2016

# Exploration of Erasure-Coded Storage Systems for High Performance, Reliability, and Inter-operability

Pradeep Subedi
*Virginia Commonwealth University*, subedip@vcu.edu

# EXPLORATION OF ERASURE-CODED STORAGE SYSTEMS FOR HIGH PERFORMANCE, RELIABILITY, AND INTER-OPERABILITY

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy at Virginia Commonwealth University.

by

PRADEEP SUBEDI

B.E., IOE Pulchowk Campus, Electronics and Communication Engineering

Director: Dr. Xubin He, Professor,

Department of Electrical and Computer Engineering

Virginia Commonwealth University

Richmond, Virginia

August, 2016

## ACKNOWLEDGEMENTS

I would like to express my deepest appreciation to my dissertation advisor, Dr. Xubin He, who has continuously supported and conveyed a spirit of adventure for performing quality research. He not only acted as an advisor but also as an unyielding source of motivation for me to push further in every step of life to achieve my goals. Without his guidance and persistent help, this dissertation would not have been possible.

I would also like to thank my committee members, Dr. Robert Klenke, Dr. Weijun Xiao, Dr. Preetam Ghosh, and Dr. Wei Cheng for helping me refine and enhance the ideas presented in this dissertation. I am constantly guided by your commitment to professional excellence and yearn to learn things from all of you.

I would also like to thank NetApp for partially funding my dissertation research. I am also grateful to Joesph Moore and Stan Skelton from NetApp for providing support and industrial expertise in developing research ideas and methodologies. All the members of the Storage Technology and Architecture Research (STAR) Lab at VCU have been very supportive of my research during the study period. I would like thank everyone in STAR Lab for the research collaboration and continuously helping me further the research objectives.

Finally, none of this would have been possible without the love, support, and patience of my family members: my parents Kabiraj Subedi and Gita Kumari Adhikari, my brother Praveen Subedi, my sister-in-law Sarita Adhikari and my dear wife Muna Bhattarai. Thank you for your never-ending support and guidance.

## TABLE OF CONTENTS


Chapter Page

Acknowledgements . . . . . ii

Table of Contents . . . . . iii

List of Tables . . . . . v

List of Figures . . . . . vi

Abstract . . . . . xi

1 Introduction . . . . . 1

1.1 Background . . . . . 4
1.1.1 Reliability and Performance of SSDs . . . . . 4
1.1.2 Reliability, Performance and Interoperability of Distributed File Systems . . . . . 5
1.2 Related Work . . . . . 8
1.2.1 Exploration of Erasure Codes . . . . . 9
1.2.2 Improving the Performance/Reliability of SSD at Storage Device Layer . . . . . 10
1.2.3 Improving the Distributed File System's Performance, Reliability, and Interoperability . . . . . 12
1.3 Problem Statement . . . . . 13
1.4 Research Approaches . . . . . 14

2 A Comprehensive Analysis of XOR-based Erasure Codes Tolerating 3 or More Concurrent Failures . . . . . 17

2.1 Erasure Codes . . . . . 17
2.1.1 Star Code . . . . . 19
2.1.2 Triple-Star Code . . . . . 20
2.1.3 Reed-Solomon-Like Code . . . . . 20
2.1.4 Cauchy-Reed-Solomon Code . . . . . 22
2.1.5 HDD1 and HDD2 Codes . . . . . 22
2.1.6 WEAVER Codes . . . . . 24

2.1.7 HoVer Codes . . . 24
2.1.8 T-Code . . . 24
2.1.9 Rabin-Like Codes . . . 26
2.1.10 Blaum-Roth Codes . . . 26
2.1.11 New Code . . . 26
2.2 Evaluation . . . 26
2.2.1 Evaluation Methodology . . . 27
2.2.2 Encoding Complexity . . . 27
2.2.3 Decoding Complexity . . . 32
2.2.4 Storage Efficiency . . . 35
2.2.5 Rebuild/Reconstruction Efficiency . . . 37
2.3 Observations . . . 40
2.4 Summary . . . 41

3 Hybrid Erasure Coded ECC scheme (EECC) for Solid State Drives . . . 45

3.1 System Design . . . 46
3.1.1 System Architecture . . . 48
3.1.2 Erasure Coding across Flash Chips . . . 51
3.2 Evaluation . . . 54
3.2.1 Performance . . . 54
3.2.2 Reliability . . . 60
3.3 Summary . . . 62

4 FINGER: A Novel Erasure Coding Scheme Using Fine Granularity Blocks to Improve Hadoop Write and Update Performance . . . 63

4.1 Motivating Example . . . 64
4.2 System Design . . . 66
4.2.1 System Architecture . . . 67
4.2.2 Block Chunking and Block-Layout . . . 68
4.2.3 Erasure Coding . . . 71
4.3 Evaluation . . . 72
4.3.1 Write-Performance . . . 73
4.3.2 Update-Performance . . . 75
4.3.3 Read and Recovery Performance . . . 78
4.4 Summary . . . 79

5 CoARC: Co-operative, Aggressive Recovery and Caching for Failures in Erasure Coded Hadoop . . . 81

5.1 A Motivating Example . . . . . 81
5.2 CoARC Design and Analysis . . . . . 84
5.2.1 Design Goals . . . . . 85
5.2.2 Design Overview . . . . . 86
5.2.3 Least Recently Failed Cache Replacement Algorithm (LRF) for Hadoop . . . . . 88
5.2.4 Analysis . . . . . 90
5.3 Experimental Evaluation . . . . . 93
5.3.1 Multi-Client Performance . . . . . 94
5.3.2 MapReduce Workloads . . . . . 97
5.4 Summary . . . . . 101

6 Conclusions . . . . . 103

References . . . . . 106

Vita . . . . . 117

## LIST OF TABLES


Table Page

I Summary on Erasure Codes Discussed in this chapter . . . . . 18

II Encoding Complexity of MDS codes tolerating 3 disk failure . . . . . 28

III Encoding Complexity of Non-MDS codes tolerating 3 disk failure . . . . . 28

IV Encoding Complexity of erasure codes tolerating $\geq 4$ disk failure . . . . . 28

V Decoding Complexity of MDS codes tolerating 3 disk failure . . . . . 33

VI Decoding Complexity of Non-MDS codes tolerating 3 disk failure . . . . . 33

VII Decoding Complexity of erasure codes tolerating $\geq 4$ disk failure . . . . . 35

VIII Storage Efficiency of erasure codes tolerating 3 disk failure . . . . . 36

IX Storage Efficiency of erasure codes tolerating $\geq 4$ disk failure . . . . . 36

X Operational latency for EECC and regular SSD . . . . . 55

XI Trace Characteristics and Baseline Average Response Time . . . . . 56

XII Corruption related statistics for different segment error rates . . . . . 57

XIII Number of program/erase cycles that can guarantee $10^{-15}$ UPER . . . . . 62

XIV Variants in LRF and Explanation . . . . . 91

## LIST OF FIGURES


| Figure | | Page |
|---|---|---|
| 1 | Encoding takes *k* data disks and encodes them on *m* coding disks. | 2 |
| 2 | Decoding takes a subset of *k*+*m* disks and recovers the *k* data disks. | 2 |
| 3 | A typical Hadoop cluster comprised of three racks, which are interconnected using 5 switches. Secondary Namenode acts only as a backup for Namenode, which keeps all metadata information about the HDFS cluster. | 6 |
| 4 | Research Approaches in this dissertation | 16 |
| 5 | Star-Code Encoding | 19 |
| 6 | Triple-Star Code Encoding | 21 |
| 7 | HDD1 Encoding $p = 5$. | 23 |
| 8 | HDD2 Encoding $p = 5$. | 23 |
| 9 | WEAVER Encoding code of $50\%$ Efficiency | 24 |
| 10 | $\text{HoVer}^{3}_{2,1}[3, 7]$ Encoding | 25 |
| 11 | T-Code Encoding $p = 7$ | 25 |
| 12 | Encoding Efficiency of MDS codes tolerating 3 disk failure | 29 |
| 13 | Encoding Efficiency of Non-MDS codes tolerating 3 disk failure | 30 |
| 14 | Encoding Efficiency of erasure codes tolerating 4 disk failure | 31 |
| 15 | Decoding Efficiency of MDS codes tolerating 3 disk failure | 32 |
| 16 | Decoding Efficiency of Non-MDS codes tolerating 3 disk failure | 34 |
| 17 | Decoding Efficiency of erasure codes tolerating 4 disk failure | 35 |
| 18 | Storage Efficiency of erasure codes tolerating 3 disk failure | 37 |

19 Storage Efficiency of erasure codes tolerating 4 disk failure . . . . . 38

20 Rebuild/Reconstruction Efficiency for Erasure codes when 1 disk fails. . . . . 39

21 Rebuild/Reconstruction Efficiency for Erasure codes when 2 disk fails. . . . . 43

22 Rebuild/Reconstruction Efficiency for Erasure codes when 3 disk fails. . . . . 44

23 Rebuild/Reconstruction Efficiency for Erasure codes when 4 data disks fail. . . 44

24 EECC system architecture showing multiple chips (each page is broken down into 8 segments and each segment protected with shorter and weaker ECC) and the HDD (log structured and storing parities for stripes formed with only one segment from each page in different flash chips) . . . . . 47

25 EECC system architecture detailing the RAID controller . . . . . 47

26 Read latency in regular SSDs employing $(33408, 32768, 40)$ BCH-code and EECC employing $(4226, 4096, 13)$ BCH-code . . . . . 49

27 Flow diagram for page read operations in EECC . . . . . 52

28 Read latency for varying fault tolerance in ECC . . . . . 56

29 Workloads average response time reduction, normalized to baseline, for varying segment error rates . . . . . 57

30 Workload's response time cumulative distribution function (CDF) comparison . 58

31 Read response time for different real-world traces . . . . . 59

32 Workload average read response time reduction for different real-world traces . 59

33 Estimated uncorrectable page error rate . . . . . 61

34 Replication vs Erasure Coding for 3 files inside a directory. . . . . 64

35 FINGER System Architecture detailing Block-chunking, RS(6,4) erasure coding and final block-layout in the Datanodes. The smaller chunks from A, B, C, and D, i.e., A1, B1, C1, and D1 are appended together to create a larger block in Datanode 1. Similarly, parities from multiple blocks, i.e., P1 from A, P2 from B, P3 from C, and P4 from D are appended together to form a larger parity block. . . . . 66

36 Write throughput for a 10GB file-write with various block-sizes. . . . . 74

37 Disk I/O for a 10GB file-write, normalized to 3-way replication. . . . . 75

38 Update throughput for updating 128MB data in a 10GB file with various block-sizes. . . . . 76

39 Disk I/O for a 128MB data update in a 10GB file, normalized to 3-way replication. 76

40 Normalized Namenode's Metadata Size for a 10GB file with various block-sizes. 77

41 Read throughput for a 10GB file-read with various block-sizes. . . . . 78

42 Recovery throughput for data reconstruction after a single node-failure. . . . . 79

43 A 10-node Hadoop cluster with 12 native blocks and 8 parity blocks. We assume the use of (10,6) erasure code and 3 nodes fail while Hadoop client is running. Blocks 1 to 6 form one data strip and Blocks 7 to 10 construct another data strip. . . . . 81

44 Amount of data read from each data block during triple node-failures. The erasure code used is RS (10,6) and block size is 64MB. . . . . 83

45 System architecture of CoARC detailing the data read and recovery process. The dotted lines with arrows represent Heartbeat messages to NameNode; the solid lines show the data/block transfer, and dotted line without arrow-ends are the new communication channels introduced to HDFS. . . . 85

46 Coalesced view of the Cache Manager and Evictor. It also details how the decision for block eviction is made. The cache in CoARC is an elastic cache, and Evictor periodically performs block eviction. . . . . 87

47 Markov Model for Failures in Erasure Coded Storage. The storage consists of $n$ nodes and employs $t$ fault tolerant erasure code. MTFF is a constant and MTRR can be computed by using Equations 5.1 and 5.2. . . . . 90

48 Total execution time in the Hadoop cluster with triple node-failures. All clients run in parallel. Failures last more than the program execution time to mimic permanent failures. . . . . 93

49 Total execution time in the Hadoop cluster with 3 node-failures. All clients run in parallel. Failures last for 1 minute only. . . . . 94

50 Total read traffic through the network in the Hadoop cluster with 3 node-failures. All clients run in parallel. Failures last more than the program execution time to mimic permanent failures. . . . . 96

51 Total read traffic through the network in the Hadoop cluster with 3 node-failures. All clients run in parallel. Failures last for 1 minute only. . . . . 97

52 Total execution time for different MapReduce applications with different failure scenarios. Failures last more than application execution time. . . . . 98

53 Total data read traffic for different MapReduce applications with different failure scenarios. Failures last more than application execution time. . . . . 99

54 Total execution time for different MapReduce applications with different failure scenarios. Failures last for 1 minute only. . . . . 100

55 Total data read traffic for different MapReduce applications with different failure scenarios. Failures last for 1 minute only. . . . . 101

## ABSTRACT

EXPLORATION OF ERASURE-CODED STORAGE SYSTEMS FOR HIGH PERFORMANCE, RELIABILITY, AND INTER-OPERABILITY

By Pradeep Subedi

A Dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy at Virginia Commonwealth University.

Virginia Commonwealth University, 2016.

Director: Dr. Xubin He, Professor,
Department of Electrical and Computer Engineering

With the unprecedented growth of data and the use of low commodity drives in local disk-based storage systems and remote cloud-based servers has increased the risk of data loss and an overall increase in the user perceived system latency. To guarantee high reliability, replication has been the most popular choice for decades, because of simplicity in data management. With the high volume of data being generated every day, the storage cost of replication is very high and is no longer a viable approach.

Erasure coding is another approach of adding redundancy in storage systems, which provides high reliability at a fraction of the cost of replication. However, the choice of erasure codes being used affects the storage efficiency, reliability, and overall system performance. At the same time, the performance and interoperability are adversely affected by the slower device components and complex central management systems and operations.

To address the problems encountered in various layers of the erasure coded storage system, in this dissertation, we explore the different aspects of storage and design several

techniques to improve the reliability, performance, and interoperability. These techniques range from the comprehensive evaluation of erasure codes, application of erasure codes for highly reliable and high-performance SSD system, to the design of new erasure coding and caching schemes for Hadoop Distributed File System, which is one of the central management systems for distributed storage. Detailed evaluation and results are also provided in this dissertation.

**Key Words**: Erasure Codes; SSD; Reliability; Performance Evaluation; Hadoop; Cloud Storage

# CHAPTER 1

# INTRODUCTION

With the scalability and wide growth of technology, a single storage system comprises of thousands of nodes or storage components. To protect the data from being lost in the face of system/device failures, various data protection mechanisms are employed. Among many techniques, *Replication* and *Erasure Coding*, are most popular [80], [15], [31], [65]. The data protection feature consumes additional storage and the additional storage is known as *storage overhead*.

Replication is a process of making multiple copies of the whole object and store these copies in different failure domain. If any data is lost, the data request can be easily served via the replica-copy. The storage overhead for replication is $100\%$ if one extra copy is maintained and $200\%$ if two copies are maintained. While this mechanism provides better availability than keeping a single copy of data, it suffers from the dreaded $200\%$ storage overhead. This large overhead becomes a major bottleneck because the recent data explosion (around $90\%$ of the total data today was created in last two years alone [6]) is generating data faster than data centers can expand to accommodate.

On the other end of the spectrum is erasure coding, which can provide more space efficient data storage while maintaining the same level of data reliability as replication [75]. An erasure coded storage system comprises an array of disks, each of which is of the same size. These disks are separated into a group of data disks ( a group of $k$ disks) and coding disks ( a group of $m$ disks). An erasure code transforms the group of $k$ disks into the group of $k+m$ disks (by performing some computation on the data in $k$ disks), such that the original data in $k$ disks can be recovered if some of the disks fail in this new group. Figure 1

demonstrates a typical encoding process and Figure 2 illustrates a decoding process. When some of the disks fail, we have a subset of the available data from the original $k$+$m$ disks. The decoding procedure operates on the data in this subset and recalculates the original data in $k$ disks. An erasure code is classified into mainly two groups Maximum Distance Separable (MDS) codes and Non-Maximum Distance Separable (Non-MDS) codes. MDS codes have a property that if any of $m$ disks fail, the original data can be reconstructed. While MDS codes have an optimal storage efficiency, they have longer reconstruction chains and thus require more data symbols to be read during fault recovery. Non-MDS codes trade storage efficiency for better read/write and reconstruction performance by creating shorter parity chains, which reduces the data read during recovery [51].

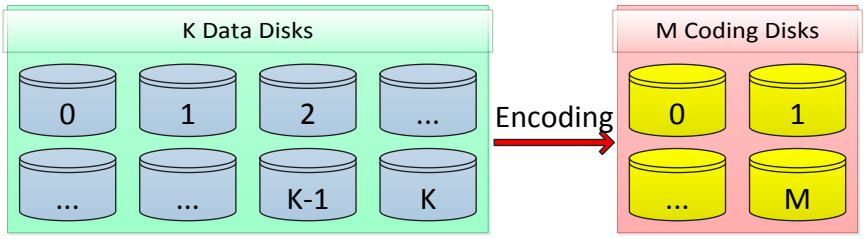



Fig. 1. Encoding takes $k$ data disks and encodes them on $m$ coding disks.

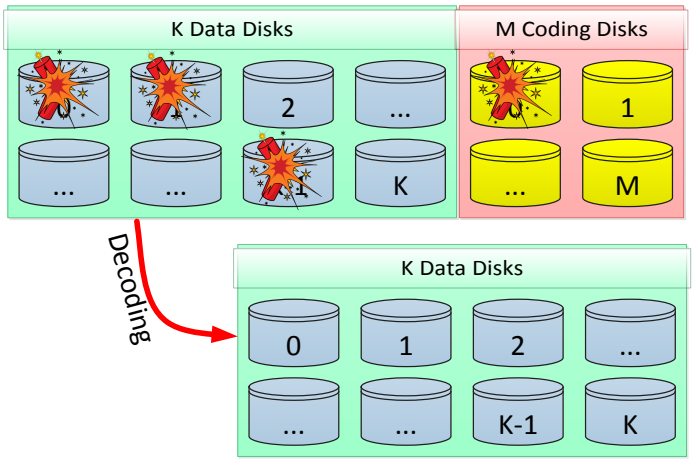



Fig. 2. Decoding takes a subset of $k$+$m$ disks and recovers the $k$ data disks.

Although erasure codes are getting popular, the naive use of a single erasure code for various application, directly affects the system performance and reliability, because of the varying encoding/decoding complexity along with the fault-tolerance among erasure codes. On the other end, just applying erasure code on the data doesn't necessarily increase the reliability and performance if the underlying storage being used is slow and unreliable. Thus, the design and use of highly reliable and superior performing storage device like SSDs should be deployed to further guarantee the user-requirement of low cost and high performance.

For low latency access and efficient data analysis, the data centers typically deploy distributed storage systems, as they are able to provide highly parallel data access and management [15], [53], [45], [34], [44]. Since the data and underlying storage is shared across multiple applications in the distributed storage environment, the management software or the file system being used should be efficient. Also, the workloads operating in the data sets can be of variable nature, i.e., some can be read-intensive, while others can be write-intensive. For the system to be high performing and interoperable, it should also be able to handle all operations like reading, writing, and updating efficiently, i.e, with low latency and low system overhead. In erasure coded storage systems, every write and update operations incur the significant overhead of reading old data and updating parity, which limits the system performance and interoperability.

Another aspect necessary for high reliability, performance and interoperability is the timely recovery of data from the failed disks/nodes/servers. If we delay the data recovery, the reliability decreases along with the increase in application latency because decoding requires the reading of surviving data blocks/chunks. Since, decoding process consumes both network and computation resource, the efficient recovery of data increases both reliability and performance. If many applications share data and if the data recovery process is not visible across all of these applications, it will also reduce interoperability of the system

because applications try to recover data independently, which increases the system latency perceived by applications.

To this end, in this dissertation, we explore various facets of erasure coded storage system and design novel techniques to improve the system performance, reliability, and interoperability.

## 1.1 Background

In this section, we present background and brief motivation to the techniques presented in this dissertation.

### 1.1.1 Reliability and Performance of SSDs

With distributed file systems garnering huge attention and the decreasing cost of SSDs, industries like NetAPP, EMC, and Google are designing and deploying huge arrays of SSDs in their production cluster. Although SSDs are getting popular because of low-power consumption, high mobility, high performance, and non-volatile characteristics, Multi-Level-Cell SSDs are less reliable.

The reliability concern regarding SSDs is addressed by Error Correcting Codes (ECC) and the SLC flash memory guarantees the desired level of reliability by using a single bit error correcting codes [69], such as Hamming Codes [37]. For ensuring the same level of reliability as SLC flash memory, a single bit ECC in an MLC NAND flash memory is insufficient. Thus MLC SSDs use ECCs that are able to tolerate multiple bit errors. The use of complex ECCs such as multi-bit error tolerating BCH code, Reed-Solomon Code or LDPC-code, increase hardware complexity and thus directly affect the read and write latency [85].

According to [47], in addition to reliability and per-bit-cost, another important metric for SSDs is read access latency, which is comprised of on-chip NAND flash memory

sensing latency, flash-to-controller data transfer latency, and ECC decoding latency. It is shown in [79] that $(34528, 32800, 108)$ BCH code takes $41.2\mu$s, whereas $(4316, 4100, 16)$ BCH code takes $5.78\mu$s to decode on the same hardware platform. As the decoding speed of ECCs is dependent upon the code-word length and bit failure tolerance [69], the use of complex ECC for better reliability adversely affects the read access latency. The ECCs are usually generated for each page and stored in the spare area of the page. If we speculate that the hardware density of flash devices is going to follow the current increasing trend, then in the near future, in order to guarantee the same level of reliability, the size of ECC will be much larger than the spare area available in the SSD. Thus, the current SSD designs are not suitable for MLC SSDs with high bit-error rate, if these high capacity SSDs are to be used in high-performance systems.

### 1.1.2 Reliability, Performance and Interoperability of Distributed File Systems

To handle petascale data volumes and offer a scalable and reliable storage platform, enterprises depend on Azure [15], GFS [31] and HDFS [65]. These storage and computing platforms use frameworks such as MapReduce [25] and Dryad [44]. A typical architecture of Hadoop cluster, which runs MapReduce framework is shown in Figure 3. The nodes/servers in each rack are connected with each other via rack-switches, which have uplinks connected to other tiers of switches providing inter-rack communication.

Hadoop uses the distributed file system HDFS [65] for reliable storage. Any file written into the HDFS is divided into fixed size *blocks*. When a client wants to write a file, it first consults the *Namenode*, which provides a list of *Datanodes*, where the client can directly write the data. The data blocks are distributed across the Hadoop cluster. The more the blocks are distributed, the better the parallelism across nodes. With the nodes being prone to failures, HDFS relies on a *replication* mechanism to guarantee reliability and data availability. By default, each block of data is replicated into 3 different data nodes. The first

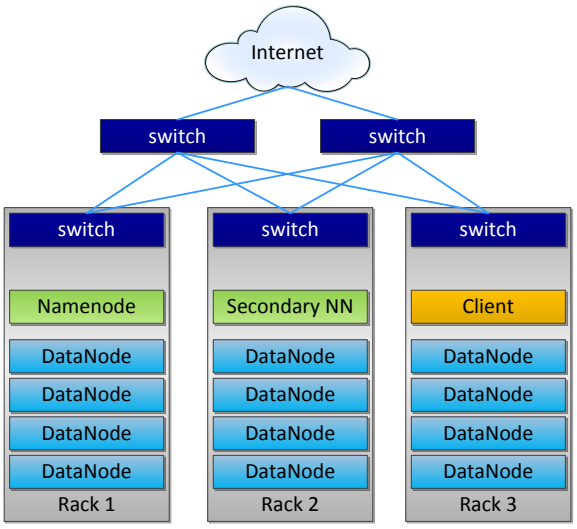



Fig. 3. A typical Hadoop cluster comprised of three racks, which are interconnected using 5 switches. Secondary Namenode acts only as a backup for Namenode, which keeps all metadata information about the HDFS cluster.

replica is placed on the local node or random node on the same rack as the client, while the second and third replicas are placed on a different rack. If there are more replicas, HDFS spreads them across the rest of the racks.

When a user uploads data to HDFS or GFS, the underlying storage layers essentially keep the data immutable, only allowing concurrent appends. While this method exploits workflow parallelism to a certain degree, it does not provide a true file system capability e.g., support for users to randomly update the contents of the files. Since HDFS does not support in-place updates, many applications which are write intensive and require file modifications need to overwrite all the file contents, even if very small changes are made, which limits the interoperability. HBASE [2] is one of the layered storage architectures, which are built on top of HDFS. A recent paper demonstrated that even if only $1\%$ of the total I/O are true HDFS writes, then $64\%$ of the final disk load is write I/O [38]; this write inflation is due to overheads such as logging and compaction, and emphasizes the need for update support in HDFS. While erasure codes can reduce the amount of data being written

to the disk for new data writes, it incurs a significant disadvantage when the data is being updated. If any one block in the encoding stripe is being updated, the parity blocks must be recomputed. To recompute parity, all other blocks in that parity stripe must be read, which incurs a significant increase in total disk I/O reducing the system performance and interoperability.

In storage systems, when any data access request cannot be completed; it is classified as a failure [63]. This failure can either be due to node unavailability or data corruption or any other I/O exceptions. Large distributed systems adopt mechanisms like timeout thresholds to distinguish between the transient failures and permanent failures because a node suffering a temporary failure will eventually recover and rejoin the network [72]. Thus, a node unavailability can be identified as a transient failure and can later translate into a permanent failure, if the node has been offline for longer than the threshold time.

When an I/O exception occurs during data access in distributed storage systems like HDFS and Azure, the client performs multiple retries and the time interval between these retries can be incremental or fixed or exponential back-off [5]. If the data block is still inaccessible, the system passes the alternate block locations (in replicated systems) or initiates degraded reads (in erasure coded systems), to fulfill the data request. In distributed systems like Hadoop, the permanent repair process is independent of the client access. If the autonomous repair process has not yet addressed the failure of the data requested by the client-read process, the client will continue to perform degraded read and throws away the reconstructed data after use. Since clients in these systems do not share the information about temporary block unavailability, all the clients accessing blocks go through the time-out approach for failure recognition, even if another client lately labeled the block as being unavailable. The lack of information sharing among various clients further aggravates the system performance because a data block recently recovered by a client during degraded read is discarded (after the read request is fulfilled) and another client accessing

the same set of data requires degraded read again. The data recovered during degraded reads are discarded because any I/O exception is classified as a transient failure, and it is naively assumed that the node will come back alive later. If these failures are classified as permanent failures later, the system needs to go through the data block repair process again. All degraded reads and node repairs require downloading data blocks from other nodes and decoding of the failed data. Thus, multiple repairs of the same block inflict a heavy burden in the distributed system's network and computation resources and reduces the interoperability among multiple clients.

Even in the case of a single client, discarding the recently recovered data blocks during degraded reads can hurt the system performance drastically. If there are multi-block failures in a strip and the client is reading the whole strip of data blocks, it only recovers the desired block, serves the request and discards the block. When the request goes to another failed block, it sees the previously recovered block as an unavailable block (even if it was recently served by degraded read). This again consumes a lot of network resources and increases the data read latency because the remaining data blocks are read multiple times during degraded read, even if they belong to the same data strip. Thus, there is a huge area for improving the reliability, performance and interoperability for erasure coded distributed file systems.

## 1.2 Related Work

To improve the performance, reliability, and interoperability of erasure coded cloud storage systems, many designs have been proposed and explored. Since we tackle the cloud storage as a broad area and tackle problems at storage device layer and management systems at the file system layer, we present some of the state-of-art research in these areas.

### 1.2.1 Exploration of Erasure Codes

It has become clear that in a storage industry, with the increase in single disk capacity with fairly constant per-bit error rate, the rate of disk failures increases. [64] concluded that even a tiny disk failure might lead to a catastrophic failure in RAID system. And among typical RAID systems, RAID-0 has a zero fault tolerance due to lack of redundancy. So implementations such as RAID-3 and RAID-5 are used to tolerate single disk failure [56]. However, with the increase in the number of disks in the storage system, the disk failure probability increases. Once a disk fails in a RAID system, there is a high probability of another disk failure in the same RAID system [56], [10], [21]. Thus, in a storage system with a large number of disks, RAID-5 is not sufficient for reliable storage [22]. Codes such as X-code [82], RDP code [22], EVENODD code [10], H-code [76], and much more have been proposed for data recovery in the case of double erasures. The need for higher fault tolerant erasure codes increases when the trends such as fairly constant disk bandwidth, with the huge increase in disk size, and use of a large number of less reliable disks in a system [22] continue. This warrants a comprehensive analysis and comparison of existing higher fault tolerant erasure codes.

Codes like Reed-Solomon [60], Triple Star [74], Star [41], and Cauchy-RS [12] support 3 or more concurrent failures. Since these codes have optimal storage efficiency, they are also known as Minimum Distance Separable (MDS) codes. While MDS codes have optimal storage efficiency, they have long reconstruction chains and thus require more data symbols to be read, during fault recovery. However, higher bandwidth and cheap storage space led to the development of Non-MDS codes such as HDD1 [71], HDD2 [71], T-Code [70], Low-complexity array codes [17], WEAVER [35], and HoVer [36] codes for efficient data recovery from multiple erasures. Non-MDS codes trade storage efficiency for better read/write and reconstruction performance [35]. As every code requires some trade-offs

in efficiency, performance, or fault tolerance, the analysis and comparison of popular erasure codes help system administrators choose even better erasure codes to optimize their systems.

Although a lot of research focuses on improving efficiency, performance or fault tolerance, to the best of our knowledge, no prior work has focused on the comprehensive analysis and comparison of all these codes.

### 1.2.2 Improving the Performance/Reliability of SSD at Storage Device Layer

A large body of work on flash-based SSD has been done in recent years. Among these, some of the works are based on modification of FTLs [48], [33], [78] to improve the lifetime and performance of an SSD while some other works [54], [45] introduce new flash file systems. Early flash file systems were designed for embedded systems. The work on the raw flash [54] and DFS [45] leveraged the virtualized flash interface, which is offered by Fusion-IO driver, to avoid physical block management complexity in traditional file systems. In this section, we discuss other research projects that aim to combine HDD and SSD to improve performance, and projects that utilize RAID concepts to improve the reliability of flash memory systems.

[20] proposes to manage the flash memory by using a self-balancing stripe scheme (SBSS). This scheme tries to improve the read performance of the system by providing parallelism across flash devices with the use of two sets of data, namely data zone and code zone. The data zone consists of original data. The popular data, which is a subset of original data, is stored in code zone by performing an exclusive-OR operation on subsets of popular data blocks. The popular data now can be served in parallel from data zone and code zone. This work stores the parities in flash memory, introducing increased writes to the flash memory storing the coded data and performs coding of only the popular data and thus will not improve the reliability of the system as a whole.

[32] improves the reliability of the flash-based memory system by employing some form of MDS codes in NAND flash chips. The caveat in this approach is that it does not try to optimize the read/write performance but just focuses on the reliability i.e. it gains reliability at the cost of sacrificed read and write performance. It also stores the redundant data in the SSD, leading to increased writes. [42] proposes to reduce the parity update overhead in RAID-5 SSD. They propose to use a partial parity cache, to store the recent parity temporarily and thus reduce the number of read operations required to calculate a parity. This scheme uses NVRAM to store the partial parity and tries to reduce the parity update time. The delayed partial parity scheme does not reduce the overall read latency of the system and it uses costly NVRAM to store redundant data. The flash-aware redundant array (FRA) scheme [49], is another approach similar to delayed partial parity. It also delays the parity block updates in the buffer memory instead of writing them directly to flash. This reduces the out-of-place update overhead of flash memory and thus alleviates the fast wear-out of flash memory. In this scheme, the parity blocks are flushed into flash memories, instead of NVRAM as in [42], while there are no requests for a while or the buffer memory is full. While [49] and [42] are targeted for reducing parity update operations.

Recently, researchers are considering the use of the SSD+HDD system for improved reliability and performance. Griffin [66] proposed the use of a log-structured HDD as a cache to absorb the write for the SSD, and then periodically migrate the data from HDD to SSD. This technique alleviates the SSD from write-intensive workloads and still retains the high performance of SSD, as all the read requests are served from the SSD. Since the HDD is a cache, not a first class partition, HDD failure will not incur data loss. Our work contrasts from this design, as our primary goal is to improve the reliability of the SSD along with read latency of the system. Griffin does not provide any improvement over the traditional read performance of SSD because it does not change the underlining SSD architecture.

I-cache [83] is an intelligently coupled array of SSD and HDD. In this technique, the SSD stores the seldom changed and most popular data block and HDD stores a log of deltas between the recently accessed I/O blocks and corresponding reference blocks in SSD. This reduces the random writes in the SSD during I/O operations. This work utilities the high-speed computation of modern CPUs to compute deltas and second, its prolonged lifetime comes from the reduced random writes to the SSD. I-cache cannot tolerate SSD-chip failure, as it relies only on the regular ECC to recover from bit-errors. This system works for mostly write intensive applications and is not read-intensive applications, which are dominant in cloud file systems.

### 1.2.3 Improving the Distributed File System's Performance, Reliability, and Interoperability

There have been extensive studies on the practicability of erasure codes in clustered storage systems [34], [7], [19], [27]. To target the needs of data-intensive applications in clustered system, Google introduced the MapReduce paradigm, which uses Google-FS as the underlying storage layer [31]. This led to the development and use of several other distributed file systems such as HDFS [65], BlobSeer [53], and MapR [4]. While HDFS and GFS support erasure codes, BlobSeer and MapR currently employ replication only. BlobSeer and MapR support in-place updates but have a high storage cost because of replication. Amazon S3 [1] provides storage service with the ability to write, read and delete an unlimited number of objects. While it supports full object reads and writes only, it does not provide fine-grained reads and writes to arbitrary offsets.

HDFS-RAID [3] offers erasure code support to HDFS to improve the storage efficiency of Hadoop File System. Since HDFS-RAID is deployed as a middle layer, the data is first replicated and later erasure coded. This causes significant consumption of network and disk bandwidth. Similar to HDFS-RAID, DiskReduce [28] also performs erasure cod-

ing in the background. DiskReduce also assumes that the data is immutable, thus providing no support for arbitrary writes and appends, which causes write amplification for appends. Zhang *et al.* [84] implement an online encoding framework for HDFS and study various MapReduce workloads. Although online encoding is done, the encoding process is strictly serial and the encoding starts only when all the blocks in the encoding stripe are available, thus limiting the performance.

Another body of work focuses on improving the performance of distributed file system during failure, i.e., improve the performance of reads issued during the degraded mode. Khan *et. al.* [46] propose a new encoding method for GF-based erasure codes to reduce the amount of data transfer during degraded reads for single disk failures. Parity declustering for recovery speedup is proposed in [81]. Zhu *et. al.* [86] build upon greedy recovery heuristic with node heterogeneity and I/O parallelism. Damakis *et. al.* [26] present regenerating codes, which minimizes the data transfer via encoding the data before transfer to other nodes for data recovery. Microsoft Azure [15] and LRC code [40] introduced extra parities to minimize the I/O during degraded reads. All of these works treat degraded reads as independent failure scenarios and do not consider that some of these unavailable/failed blocks might be accessed in near future, which will lead to degradation in the read performance of the distributed storage.

## 1.3 Problem Statement

In this dissertation, we state our research purpose as *to explore the erasure-coded storage systems for achieving reliable, interoperable, and high-performance storage. We propose to explore multiple layers of the erasure coded storage for gain in reliability, performance, and interoperability. We explore, analyze and comprehensively evaluate high fault tolerant erasure codes to provide research insights. We further delve deeper into the system storage layer and design new erasure-coded ECC design for SSDs to improve the overall*

*system performance and reliability. We then design new erasure coding techniques and data layout mechanisms for distributed file systems like Hadoop and also design highly efficient data recovery approach for distributed storage systems, where data is shared across multiple application, which leads to increase in performance and interoperability among various clients.*

## 1.4 Research Approaches

In this dissertation, we explore various layers of an erasure coded storage system, and propose following insights and techniques for high performance, reliable and interoperable storage system:

In large-scale distributed storage systems, RAID systems are used to recover from multiple disk/node failures. Although previous researchers focus on improving efficiency, performance or fault tolerance, we first explore the impacts of choice of various erasure codes. The aim of this work to help better understand the current erasure codes in RAID systems and point out open issues that can be subject to further research. We focus on encoding/decoding efficiency, storage efficiency and rebuild/reconstruction efficiency of erasure codes for comparison.

We apply the concept of erasure code in Hybrid Erasure Coded (**EECC**) Scheme for Solid State Drives for achieving a gain in both Performance and Reliability. In EECC, instead of using complex ECC schemes to recover from bit errors, we propose to employ weak-ECC to improve read performance and then bolster the reliability by using erasure codes. This design focuses on read-intensive applications, where read performance is of higher importance and the writes are very few. Since the ECC decoding speed is dependent upon the code word length and number of tolerable bit failures, weak-ECC provides faster decoding speed. The read performance is further improved by overlapping the flash to controller data transfer of one-page segment with the decoding of another page segment.

This design uses a spare HDD to store parities so that writes to parities do not aggravate the SSD lifetime.

We design a Fine Granularity Erasure Coding (**FINGER**) for Hadoop Distributed File System to improve the write and update throughput of erasure coded storage. FINGER is a new block-layout algorithm for erasure-coded Hadoop, which eliminates the parity block re-computation when full data-block updates are performed by the system client. The main idea is to perform erasure coding within a block instead of across the blocks and design the block-layout so that no extra metadata information is needed. We also add the block-update feature in Hadoop File System for better interoperability.

Finally, based on the insights obtained from the comprehensive evaluation of erasure codes and complete understanding of the distributed file system, we propose **CoARC** (Co-operative, Aggressive Recovery and Caching), which is a new data-recovery mechanism for unavailable data during degraded reads in distributed cloud file systems. We identify that the lack of data sharing during temporary failures and retrieval of just the requested data block place heavy burden/overhead on the system's network resources and increases the job execution time. The main idea of CoARC is to recover not only the data block that was requested but also other temporarily unavailable blocks in the same strip and cache them in a separate data node. The caching enables the sharing of the recovered data between multiple clients and increases the data reliability of the system. We also propose an LRF (Least Recently Failed) cache replacement algorithm for such a kind of recovery caches. CoARC achieves the goal of improving reliability, performance and interoperability for distributed erasure coded file system.

Figure 4 shows the research approaches presented in this dissertation. The system under consideration consists of N storage racks. These storage racks/servers consists of SSDs and HDDs and the data layout is managed by central manager i.e, Hadoop Distributed File System. The blocks in yellow are the ideas presented in following chapters. The rest

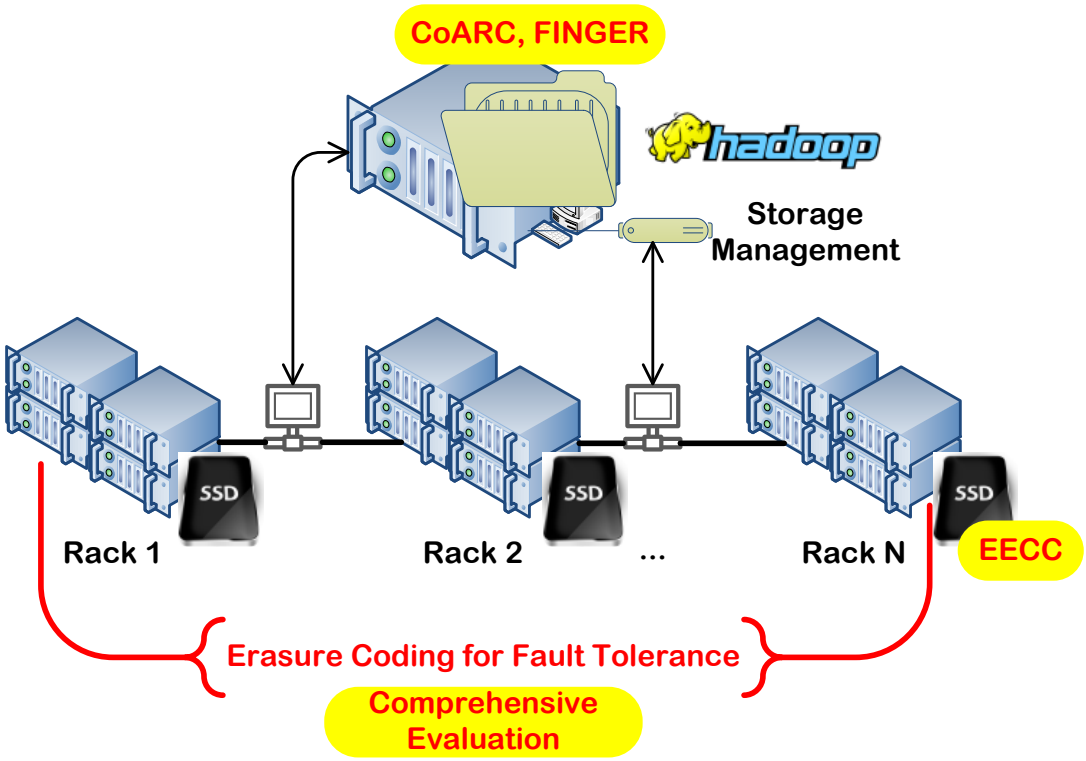



Fig. 4. Research Approaches in this dissertation

of the dissertation is organized as follows. We discuss the exploration on erasure codes in Chapter 2. Chapter 3 details the EECC design and Chapter 4 describes how to improve the write and update performance of Hadoop file system using FINGER. CoARC mechanism is explained in Chapter 5 and Chapter 6 concludes this dissertation.

# CHAPTER 2

# A COMPREHENSIVE ANALYSIS OF XOR-BASED ERASURE CODES TOLERATING 3 OR MORE CONCURRENT FAILURES

As information technology is developing at a rapid rate, it is a must to have a reliable storage system. With the increase in data size and computation capacity, we prefer storage systems, that have larger capacity, are persistent and more reliable. [80] pointed out that it is one of the most important performance metrics in storage systems. Disk Arrays [56], such as Redundant Array of Inexpensive Disks (RAID) have been a popular choice inside companies, universities, and organizations because of their ability to ensure data reliability, integrity and availability.

## 2.1 Erasure Codes

In this section, we give a brief introduction to each of the Erasure Codes presented in this chapter. Table I gives a general idea about these codes.

Table I. Summary on Erasure Codes Discussed in this chapter

| Name | MDS | | | | Non-MDS | | | |
|---|---|---|---|---|---|---|---|---|
| | 3 Erasures | | $\geq 4$ Erasures | | 3 Erasures | | $\geq 4$ Erasures | |
| | Clustered | Random | Clustered | Random | Clustered | Random | Clustered | Random |
| Star [41] | $\surd$ | $\surd$ | | | | | | |
| Triple-star [74] | $\surd$ | $\surd$ | | | | | | |
| RS-Like [29] | $\surd$ | $\surd$ | | | | | | |
| Cauchy-RS [12] | $\surd$ | $\surd$ | $\surd$ | $\surd$ | | | | |
| HDD1 [71] | $\surd$ | $\surd$ | | | | | | |
| HDD2 [71] | | | | | $\surd$ | $\surd$ | | |
| WEAVER [35] | | | | | $\surd$ | $\surd$ | $\surd$ | $\surd$ |
| HoVer [36] | | | | | $\surd$ | $\surd$ | $= 4$ *erasures* | $= 4$ *erasures* |
| T-Code [70] | | | | | $\surd$ | $\surd$ | $\surd$ | $\surd$ |
| Rabin-Like [30] | | | $\surd$ | $\surd$ | | | | |
| Blaum-Roth [11] | $\surd$ | $\surd$ | $\surd$ | $\surd$ | | | | |
| New Code [17] | | | | | $\surd$ | | $= 4$ *erasures* | |

### 2.1.1 Star Code

We first briefly describe Star Code [41] which tolerates up to three concurrent failures. It is an extension of the EVENODD code [10], which was initially proposed to address double failures in a disk array system. Since the data from multiple disks form a multi-dimensional array, each disk failure corresponds to a column erasure. This code uses three parity disks and $p$ information disks. Here the value of $p$ being a prime number does not limit its use in the practical system, with the use of a simple technique called codeword shortening [51].

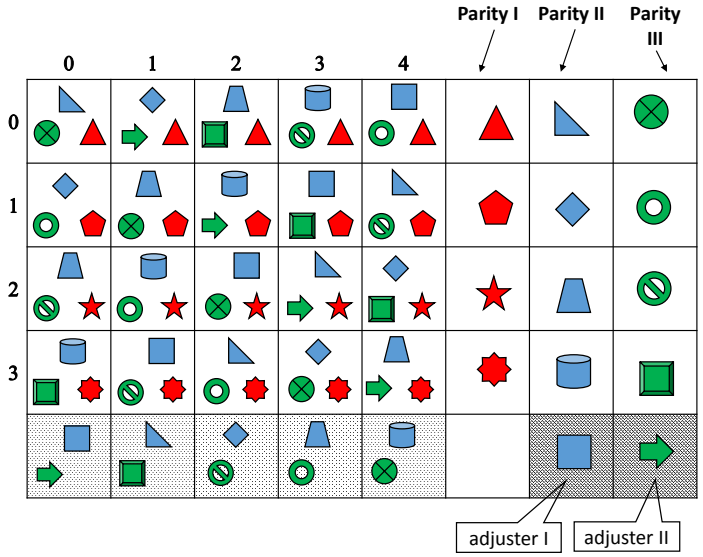



Fig. 5. Star-Code Encoding

The encoding process consists of a $(p-1) \times (p+3)$ array, where the first $p$ columns are information columns, and the last 3 columns are parity columns. In column $p$, a parity symbol is computed as the XOR of all information symbols in the same row. The array is now augmented with an imaginary row $p-1$ for the computation of parity symbols in columns $p+1$ and $p+2$. The imaginary row symbols are assigned zero values and all the information symbols along the diagonal of $slope1$ is XORed and the symbol $p-1$ in $p+1$

column becomes a non-zero symbol, called $adjuster$. Now, this adjuster is removed from the array by performing the XOR addition of the adjuster to all symbols in column $p + 1$. For the last parity column, similar operation as the column $p + 1$ is performed. However, the information symbols along the diagonal of $slope - 1$ are used to compute the parity symbols. Figure 5 shows the encoding procedure of star-code for $p = 5$. For the decoding algorithm and complete understanding of the Star Code, please refer to [41].

### 2.1.2 Triple-Star Code

The Triple-Star code is an extension of Rotary Code [73]. This code uses $p - 1$ information columns and $3$ parity columns to recover from three random disk failures; i.e., the Triple-Star coded array size is $(p - 1) \times (p + 1)$. In this code, the array is augmented with an imaginary row $0$, where the symbols are assigned value $zero$. The column $p - 1$ consists of Horizontal parity symbols (computed as the XOR sum of all information symbols in the same row). Triple-Star code computes the XOR sum of information symbols along the same diagonal of $slope - 1$ for the parity symbols in column $p$ and this XOR sum also includes the parity symbols in column $p - 1$. The computation of parity symbols in the last column is similar to column $p - 1$, the difference is in the slope of the diagonal (chosen as $1$). The encoding procedure is illustrated in Figure 6. We can see the improvement in this code from Star code because it does not use $adjusters$ in the encoding/decoding operations, thus improving efficiency [74]. We do not describe the decoding algorithm and if interested, please refer [74].

### 2.1.3 Reed-Solomon-Like Code

The Reed-Solomon-Like (RS-Like) codes [29] are based on circular permutation matrices (CPM) and Reed-Solomon code [60]. This binary MDS array code is a class of binary linear code, where information bits form a $m \times n$ array and parity bits form an $m \times n$

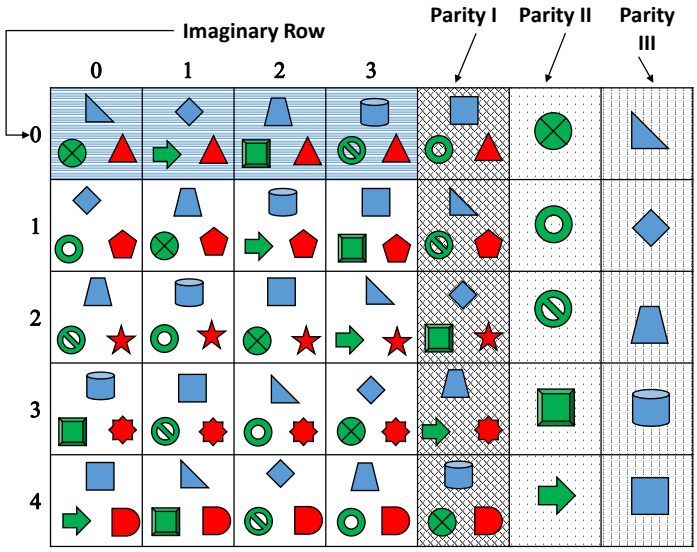



Fig. 6. Triple-Star Code Encoding

array, where $m + 1$ is a prime integer and $n$ is the information disk. The Reed-Solomon Code is based on the Vender-monde matrix:

$$H_V = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ x_1 & x_2 & x_3 & \cdots & x_n \\ x_1^2 & x_2^2 & x_3^2 & \cdots & x_n^2 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_1^{r-1} & x_2^{r-1} & x_3^{r-1} & \cdots & x_n^{r-1} \end{bmatrix}$$

where, $x_i$s for $1 \leq i \leq n$, are distinct from each other. Reed-Solomon code uses $H_V$ as a parity-check matrix. The minimum distance of the code is $r + 1$. The RS-like code is based on extended Reed-Solomon Code, which is based on the following parity check matrix (for $r = 3$):

$$H_{EV} = \begin{bmatrix} 1 & 0 & 0 & 1 & 1 & 1 & \cdots & 1 \\ 0 & 1 & 0 & x_1 & x_2 & x_3 & \cdots & x_n \\ 0 & 0 & 1 & x_1^2 & x_2^2 & x_3^2 & \cdots & x_n^2 \end{bmatrix}$$

[29] uses CPM matrix to extend the Reed-Solomon Code (for details refer [29]). When $r$ = 2, this code is the same as the EVENODD code.

### 2.1.4 Cauchy-Reed-Solomon Code

Cauchy-Reed-Solomon code [12] is also based on the Reed-Solomon codes. The encoding of CRS code is a matrix vector multiplication with the help of a special generator matrix built on Vandermonde matrices [67]. The Cauchy RS coding modifies the Galois-field operations over $GF(2)$ to bit-selection, and subsequent XOR operations are faster. This leads to an increase in the efficiency because the standard RS algorithm [57] consumes most of the time in Galois-field operations over finite fields.

### 2.1.5 HDD1 and HDD2 Codes

HDD1 [71] is a MDS code and HDD2 [71] is Non-MDS code designed to tolerate triple disk failures in RAID systems. In HDD1 scheme, $p + 1$ disks are used, where each disk consists of $p - 1$ logical blocks ($p$ is a prime number). The encoding of the HDD1 code is illustrated in Figure 7. Instead of placing the horizontal and anti-diagonal parities, the parities are distributed across disks (corresponds to each column in the disk array). The parities are calculated by evaluating the XOR sum of all information symbols along diagonals of $slope\ 0$, $slope\ 1$ and $slope\ -1$ for horizontal, anti-diagonal and diagonal parities respectively. The horizontal and anti-diagonal parities include all data rows and data columns, but the diagonal parities leave some data elements and include anti-diagonal parity elements.

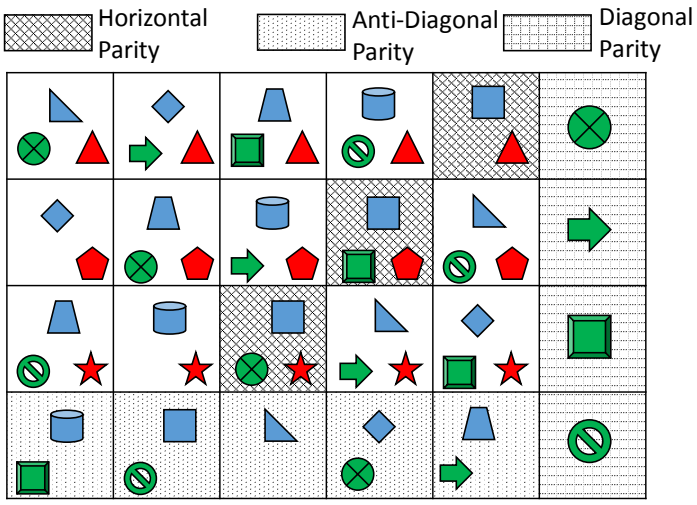



Fig. 7. HDD1 Encoding $p = 5$.

HDD2 scheme utilizes a similar encoding procedure as HDD1 scheme, the difference lies in the placement of the parity elements and the addition of one more disk (column in the disk array); i.e., the disk array is represented as a $(p-1) \times (p+1)$ matrix. Also, all data elements are involved in the computation of diagonal parity elements. Figure 8, illustrates the encoding algorithm. The details about the decoding algorithm can be found in [71].

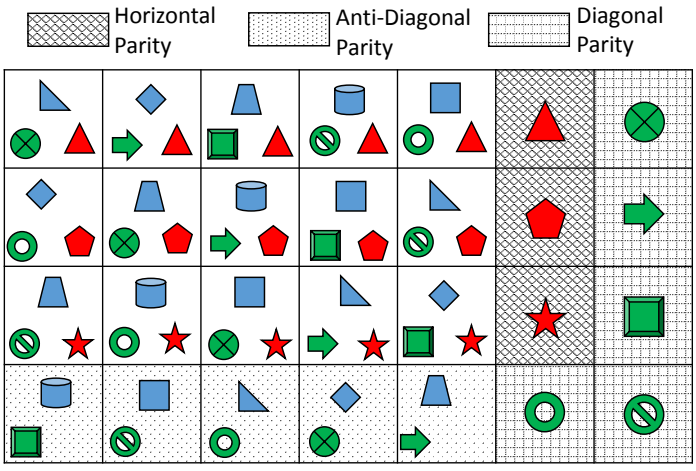



Fig. 8. HDD2 Encoding $p = 5$.

### 2.1.6 WEAVER Codes

WEAVER codes [35] are highly fault tolerant codes (support up to 12 disk failures). These codes are in general Non-MDS vertical codes. The number of data elements contributing to the computation of parity elements is fixed, i.e., is independent of stripe size. There are various configurations for WEAVER codes, which tradeoffs between storage efficiency, fault tolerance and encoding and decoding complexity. In this chapter, we consider a typical case of WEAVER code. The encoding procedure is shown in Figure 9.

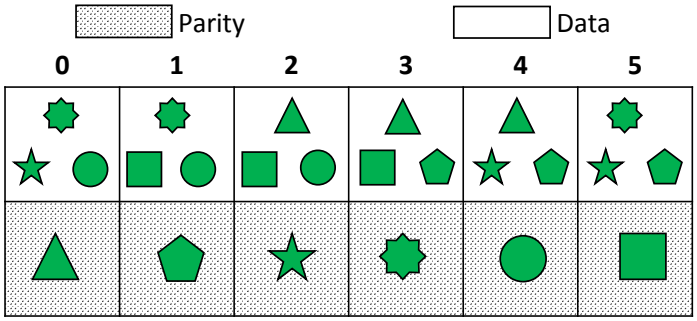



Fig. 9. WEAVER Encoding code of $50\%$ Efficiency

### 2.1.7 HoVer Codes

HoVer codes [36] are approximately MDS codes. A typical HoVer code supporting $3$ fault tolerance consists of up-diagonal, down-diagonal and horizontal parity elements and these codes were mainly designed for better reconstruction performance. Figure 10 shows an example of HoVer code. In HoVer codes some of the elements are spare, which are neither data nor parity elements.

### 2.1.8 T-Code

T-Codes [70] are Non-MDS codes that can tolerate up to 15 disk failures. The encoding of T-Code is shown in a typical case in Figure 11. The elements in the shaded boxes

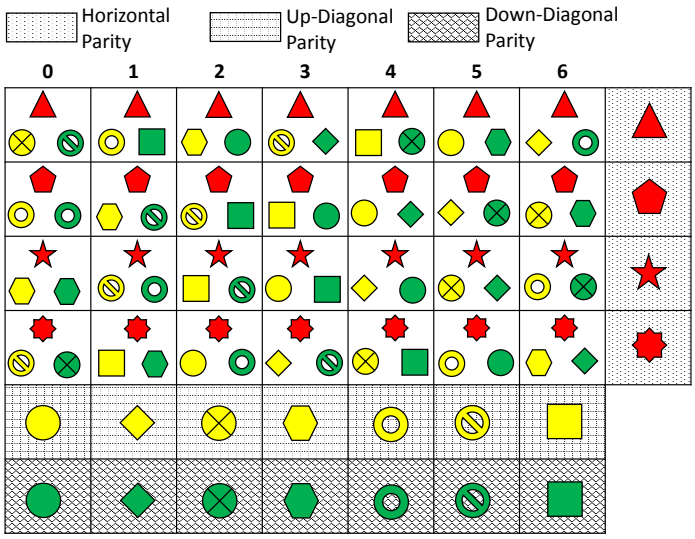



Fig. 10. HoVer$^{3}_{2,1}[3, 7]$ Encoding

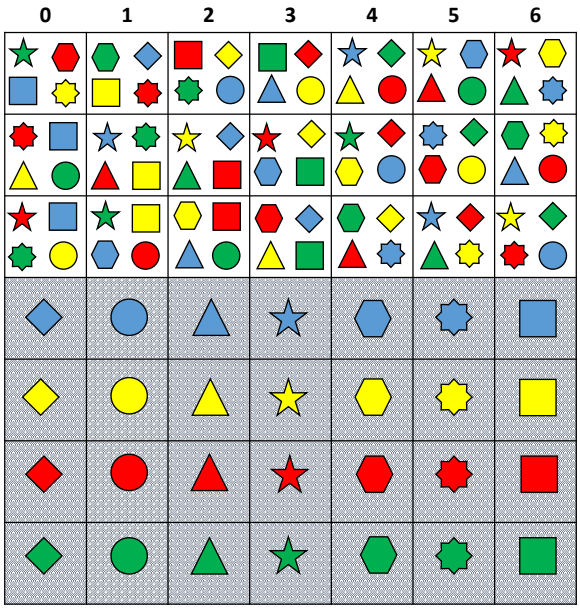



Fig. 11. T-Code Encoding $p = 7$

are parity elements and other are data elements, and this configuration can tolerate three disk failures. In some cases, T-code encoding does not support specified fault tolerance, so readers are encouraged to refer to [70] to look at the parameters (disk configurations) that

can be used to construct up to 15 fault tolerance T-code.

### 2.1.9 Rabin-Like Codes

Rabin-Like Codes [30] are MDS codes that are based on Rabin codes [58]. Rabin Codes are based on Cauchy Matrix. The Rabin codes are extended based on Circulation Permutation Matrices like RS-Like codes. These codes tolerate four or more than four concurrent failures.

### 2.1.10 Blaum-Roth Codes

Blaum-Roth Codes [11, 9] are MDS codes that support more than 3 disk failures. These codes are similar to Reed-Solomon codes, except that they are defined over certain *polynomial rings* rather than over Galois Fields. Since these codes are defined over polynomial rings, the multiplications are reduced to cyclic shifts of binary vectors, and arithmetic ring operations require only XOR operations, these codes are easier to implement than RS codes from a hardware point of view.

### 2.1.11 New Code

New Code [17] is an array code designed to tolerate four clustered disk failures. This code tolerates all combinations of cluster erasures that fall into three and fewer clusters and like other codes mentioned above only involve XOR operations.

## 2.2 Evaluation

In this section, we evaluate the encoding/decoding efficiency, storage efficiency, and rebuild/reconstruction efficiency for the erasure codes mentioned in the Section 2.1.

### 2.2.1 Evaluation Methodology

We evaluate all the codes mentioned in the Section 2.1 except New Codes because the New Codes only target clustered erasures and do not support random erasures. We have grouped erasure codes into three groups: MDS codes tolerating $3$ disk failures, Non-MDS codes tolerating $3$ disk failures, and Codes tolerating $\geq 4$ disk failures. The codes tolerating $\geq 4$ disk failures contain both MDS and Non-MDS codes.

### 2.2.2 Encoding Complexity

In this sub-section, we compare the encoding efficiency of various erasure codes. Encoding efficiency is defined as

$$\textit{Encoding Efficiency} = \frac{\textit{Total XOR operations in Encoding}}{\textit{Number of data elements}} \tag{2.1}$$

The XOR operations in Equation 2.1 refers to the total number of XOR operations required for computing all the parity elements and the denominator excludes the parity elements in the disk array.

In the subsequent discussions, parameter $p$ is a prime number, $r$ is the number of redundant/parity disks and $n$ is the total number disks. Table II shows Total XOR operations and number of data elements in each erasure codes. For Blaum-Roth code, the encoding complexity is $rpn + \frac{7}{2}r^2p - \frac{5}{2}rp - 2r^2 + 2r$. For fair comparison, we consider $r$ = 3 and $n$ = $p$. For Cauchy-RS code, the encoding complexity is $(n-r)rL^2$ where, $L$ is a message packet size, and $(n-r) \leq 2^L$.

In Figure 12, the code with better efficiency has a lower value because the efficient code minimizes the total number of XOR operations. We can observe that the efficiency of Cauchy-RS degrades with the increase in the number of disks, the reason is due to the parameter $L$. If we want an increase in the number of disks, the value of $L$ also increases, and

Table II. Encoding Complexity of MDS codes tolerating 3 disk failure

| Erasure Codes | Number of XOR Operations | Number of Data Elements |
|---|---|---|
| Star Code | $3p^2 - 7p$ | $(p-1) \times (p-1)$ |
| Triple-Star Code | $3p^2 - 9p + 6$ | $(p-1) \times (p-1)$ |
| RS-Like Code | $3p(p-1)$ | $p \times (p-1)$ |
| Blaum-Roth Code | $3p^2 + 24p - 12$ | $(p-1) \times (p-3)$ |
| Cauchy-RS Code | $3(p-3)L^2$ | $L \times (p-3)$ |
| HDD1 | $3p^2 - 10p + 6$ | $(p-2) \times (p-1)$ |

Table III. Encoding Complexity of Non-MDS codes tolerating 3 disk failure

| Erasure Codes | Number of XOR Operations | Number of Data Elements |
|---|---|---|
| HDD2 | $3p^2 - 8p + 2$ | $(p-2) \times p$ |
| WEAVER | $2p$ | $p$ |
| HoVer | $3p^2 - 12p + 3$ | $p \times (p-3)$ |

Table IV. Encoding Complexity of erasure codes tolerating $\geq 4$ disk failure

| Erasure Codes | Number of XOR Operations | Number of Data Elements |
|---|---|---|
| Rabin-Like | $(2r+1)(p^2 - p)$ | $(p-1) \times p$ |
| Blaum-Roth | $rp^2 + \frac{7}{2}r^2p - \frac{5}{2}rp - 2r^2 + 2r$ | $(p-1) \times (p-r)$ |
| Cauchy-RS | $r(p-r)L^2$ | $L \times (p-r)$ |
| WEAVER | $(r-1)p$ | $p$ |

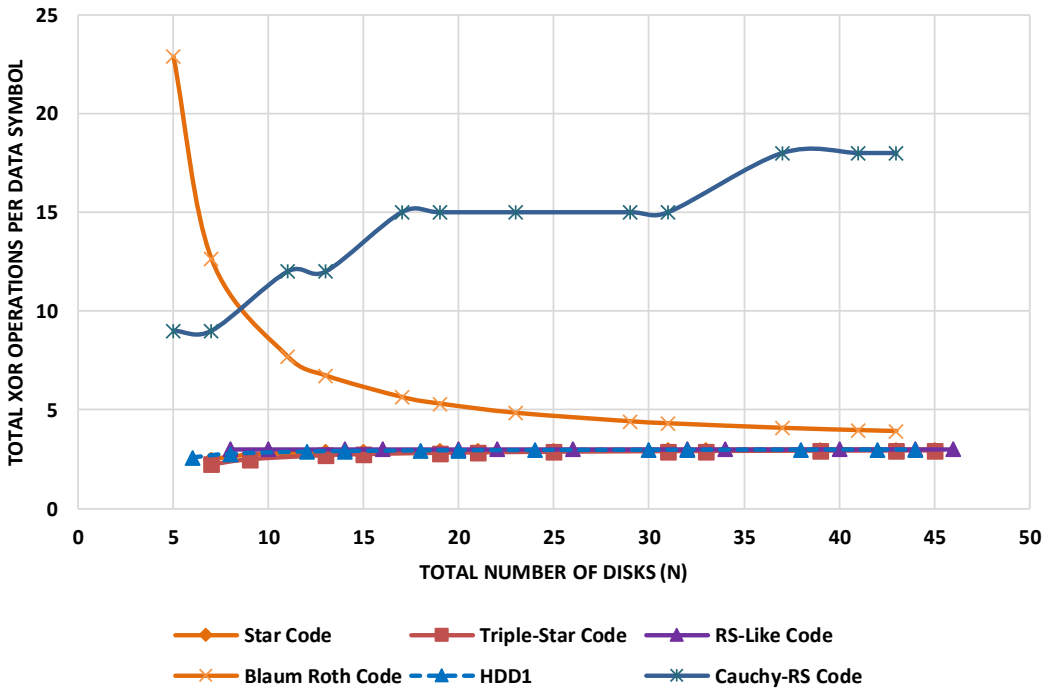



Fig. 12. Encoding Efficiency of MDS codes tolerating 3 disk failure

this leads to increase in XOR operations, reducing the encoding efficiency. The complexity of Star Code, Triple-Star Code and RS-Like Code remains fairly constant (around 3 XORS per data symbol), but Triple-Star Code has improvement in complexity when compared with Star and RS-Like Code (refer Table II). The complexity of Blaum-Roth code is fairly high for small values of $N$ but approaches 3 asymptotically.

Table III shows the encoding complexity of Non-MDS Codes and we only consider WEAVER codes of three fault tolerance. For HoVer code, we consider a typical case of HoVer$^3_{2,1}[p-3,p]$. The encoding complexity of WEAVER code is $nq(k-1)$, where, $q$ is the number of logical stripes containing parity elements and $k$ is the parity-in degree (number of data elements contributing to the construction of a parity element), which is independent of stripe size. As the construction of WEAVER code directly affects storage efficiency, we simply consider the WEAVER codes of highest storage efficiency, i.e., $50\%$

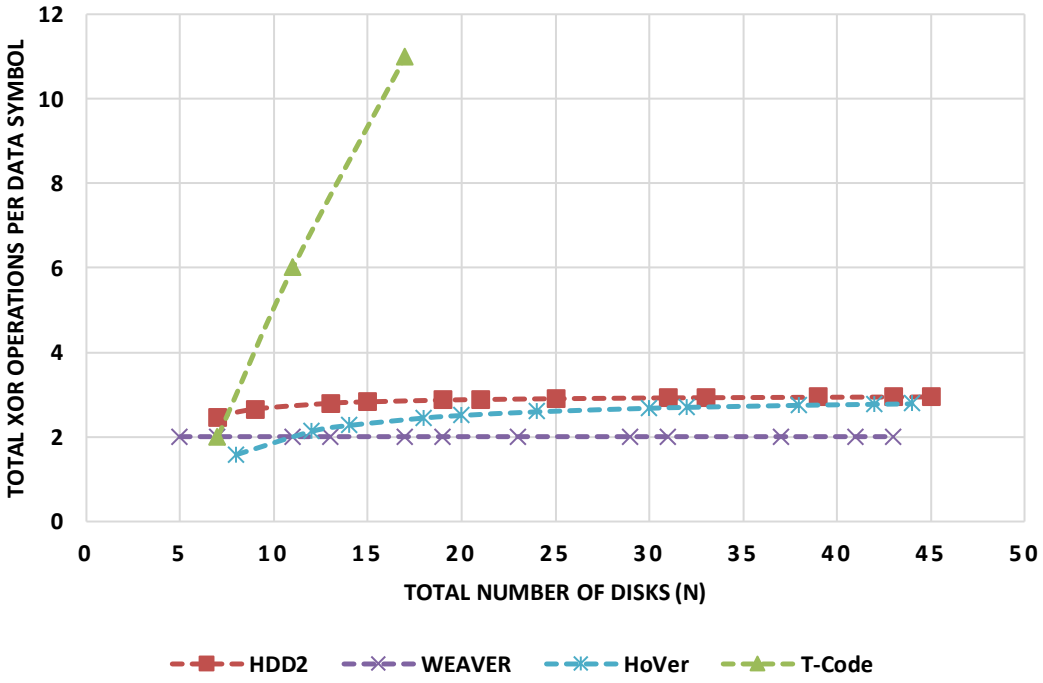



Fig. 13. Encoding Efficiency of Non-MDS codes tolerating 3 disk failure

where $k$ = *fault tolerance* = 3. The encoding complexity of T-code is $mnt$, where $m$ is the number of data rows, $n$ is the total number of disks and $t$ is fault tolerance. Since all forms of T-code are not three fault tolerant, we just plot the values in Figure 13 for some valid T-codes tolerating three disk failures.

From Table III and Figure 13, we can observe that the typical case of WEAVER ($50\%$ storage efficiency) is the most efficient code (in terms of encoding). The encoding complexity remains constant at 2 because the parity-in degree is independent of the stripe size. The complexity is fairly low of HoVer code for fewer number of disks but approaches 3 asymptotically as $n$ increases.

In Table IV, we show the encoding complexity for codes that tolerate $\geq 4$ simultaneous failures and $r$ represents the number of redundant/parity disks and is equivalent to fault tolerance. Since HoVer codes tolerate at most $4$ concurrent failures, for fair comparison, we

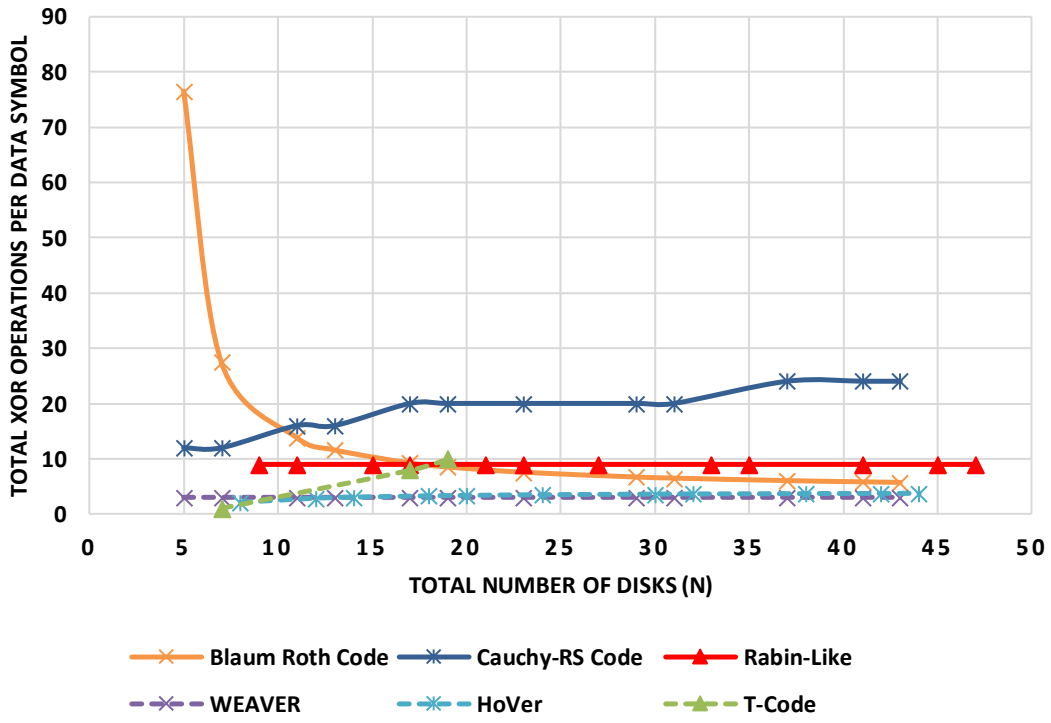



Fig. 14. Encoding Efficiency of erasure codes tolerating 4 disk failure

consider only $4$ concurrent failures in Figure 14. The encoding complexity of Rabin-Like code, Blaum-Roth Code, Cauchy-RS code, WEAVER code, and HoVer code are $9p^2 - 9p$, $4p^2 + 46p - 24$, $4(p-4)L^2$, $3p$, and $4p^2 - 16p + 3$ respectively. Similar to Figure 13, we only show the encoding efficiency for valid combinations of T-Code in Figure 14. We see that, in the case of $4$ fault tolerance, for a low number of disks, Blaum-Roth code has least encoding efficiency and as disks increase, the encoding complexity goes to $5$ asymptotically. The complexity of Cauchy-RS code is fairly large for all values of $n$. Similar to Figure 13, WEAVER has lowest encoding complexity and remains constant at 3 XORs per data symbol.

### 2.2.3 Decoding Complexity

In this sub-section, we compare the decoding efficiency of various erasure codes. Decoding efficiency is defined as

$$Decoding\ Efficiency = \frac{Total\ XOR\ operations\ in\ Decoding}{Number\ of\ data\ elements} \tag{2.2}$$

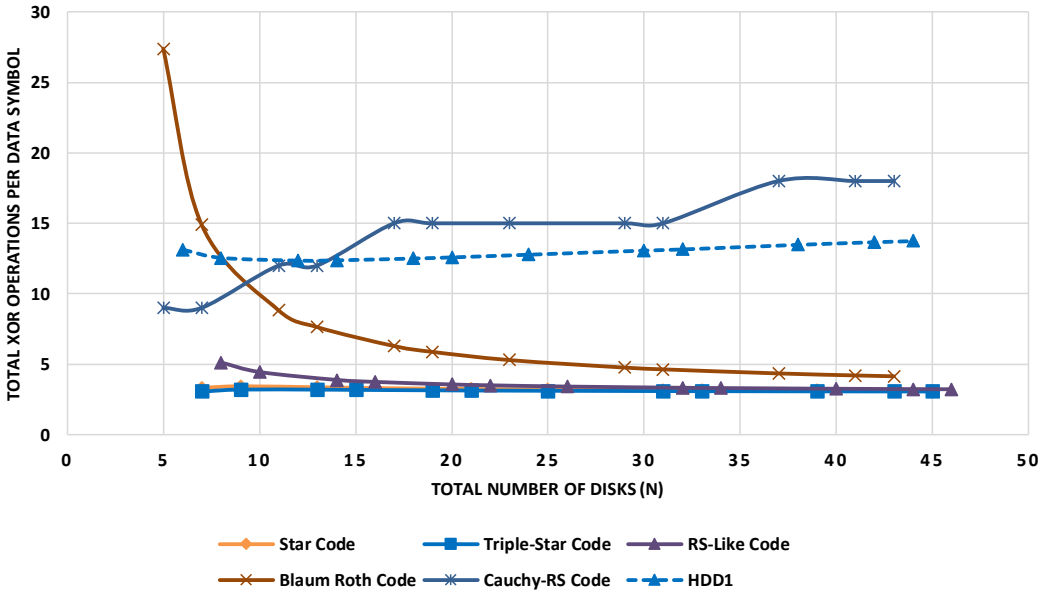



Fig. 15. Decoding Efficiency of MDS codes tolerating 3 disk failure

The total XOR operations in Equation 2.2 refer the XORs required to recover all the data elements in case of disk failure. We do not consider the recovery of parity in this case.

The XOR operations in Star code decoding include two additional parameters, decided by erasure pattern. So, we cannot compare the results from [41] directly with other codes. Thus, we count XOR operations required in each step of the algorithm. The decoding algorithm of HDD1 and HDD2 involve Gaussian Elimination, and as it is very hard to calculate the exact number of XORs that are required during the Gaussian elimination process [71] simulates a number of cases and calculates the average value to estimate the

Table V. Decoding Complexity of MDS codes tolerating 3 disk failure

| Erasure Codes | Number of XOR Operations | Number of Data Elements |
|---|---|---|
| Star Code | $3p^2 - p - 17$ | $(p-1) \times (p-1)$ |
| Triple-Star Code | $3p^2 - 3p - 11$ | $(p-1) \times (p-1)$ |
| RS-Like Code | $3p^2 + 6p - 3$ | $p \times (p-1)$ |
| Blaum-Roth Code | $3p^2 + 33p - 21$ | $(p-1) \times (p-3)$ |
| Cauchy-RS Code | $3(p-3)L^2$ | $L \times (p-3)$ |
| HDD1 | $7.5p^2 - 25.5p + 18 + Gaus$ | $(p-2) \times (p-1)$ |

Table VI. Decoding Complexity of Non-MDS codes tolerating 3 disk failure

| Erasure Codes | Number of XOR Operations | Number of Data Elements |
|---|---|---|
| HDD2 | $7.5p^2 - 20.5p + 15 + Gaus$ | $(p-2) \times p$ |
| WEAVER | $2p$ | $p$ |

XORs required in this step of the algorithm. We use this value as provided in our analysis, and $Gaus$ in Tables V and VI refers to this value. Table 2.2 lists the Total XORs required to recover data, when three disks fail in a RAID system using the listed erasure codes. The decoding efficiency in Figure 15 also follows the same pattern as encoding efficiency in 2.2.2. Among MDS codes supporting three disk failures, Triple-Star code has the highest decoding efficiency, and the Cauchy-RS has the lowest performance in terms of decoding complexity. Although the decoding complexity of HDD1 remains fairly constant with an increase in the number of disks, it is outperformed by other MDS codes except Cauchy-RS.

As the HoVer code decoding algorithm is very complex and depends upon the location of the erasures, we do not provide a mathematical equation for the HoVer code decoding algorithm in Table VI. The decoding complexity of T-code is unsure but lies between $3m^2$ and $3m^2$. So, we run simulations for HoVer and T-code and average the results from simu-

lations for a specific configuration. Since the values were too-large for T-code, we do not include them in Figure 16.The decoding complexity of HoVer increases significantly with an increase in the number of disks because in some cases it needed to read almost all data elements to reconstruct the lost data. The HDD2 has slight performance improvement than the HDD1, and the WEAVER has best decoding efficiency.

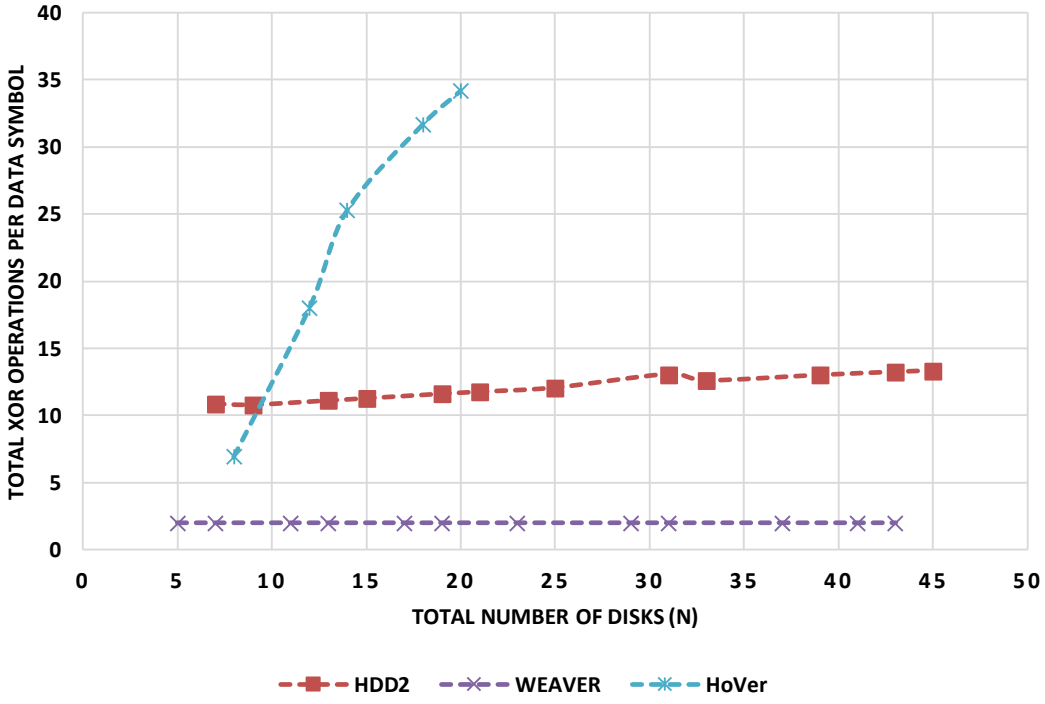



Fig. 16. Decoding Efficiency of Non-MDS codes tolerating 3 disk failure

Table VII shows the decoding complexity for erasure codes tolerating $\geq 4$ concurrent failures. For $4$ simultaneous failures, the decoding complexity of Rabin-Like code, Blaum-Roth code, Cauchy-RS code, and WEAVER codes are $9p^2 + 86p$, $4p^2 + 62p - 28$, $4(p - 4)L^2$, and $3p$ respectively. As the complexity of HoVer code decoding increases with an increase in the number of erasures and has high decoding complexity (Figure 16), we do not compare it with other codes for the case $4$ erasures. From Figure 17, we can conclude that, if the number of disks is low, and if we want fast decoding, it is better not to use

Table VII. Decoding Complexity of erasure codes tolerating $\geq 4$ disk failure

| Erasure Codes | Number of XOR Operations | Number of Data Elements |
|---|---|---|
| Rabin-Like | $O((p-1)^3 r^4)$ | $(p-1) \times p$ |
| Blaum-Roth | $rp^2 + (3r^2 + \frac{1}{2}r)p + r^2 - \frac{1}{2}r$ | $(p-1) \times (p-r)$ |
| Cauchy-RS | $r(p-r)L^2$ | $L \times (p-r)$ |
| WEAVER | $(r-1)p$ | $p$ |

Blaum-Roth code. Cauchy-RS code has least decoding efficiency with an increase in the number of disks, and WEAVER outperforms other codes in decoding efficiency.

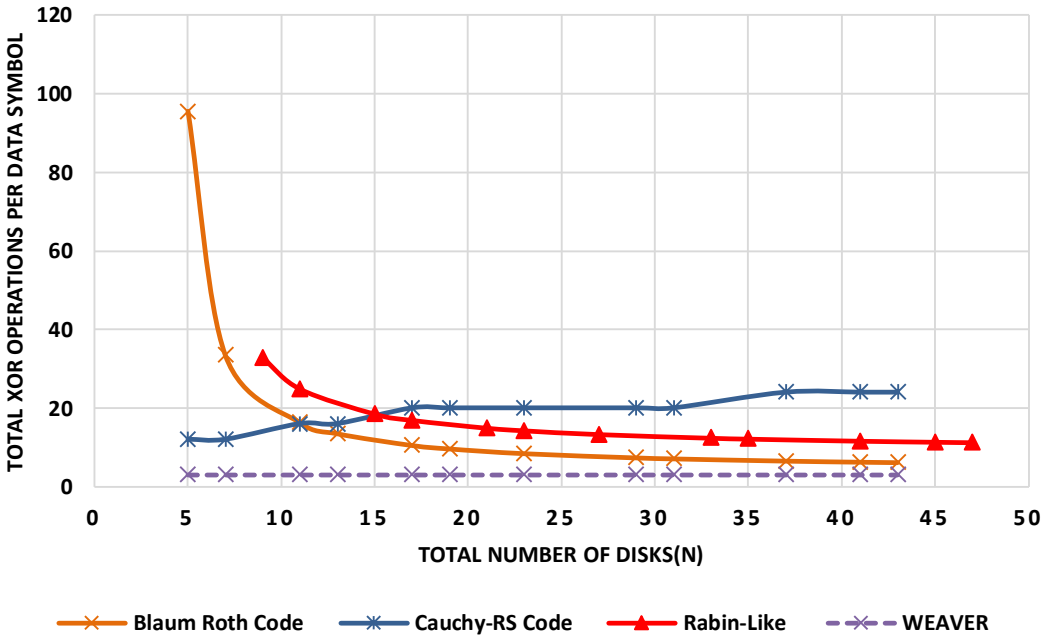



Fig. 17. Decoding Efficiency of erasure codes tolerating 4 disk failure

### 2.2.4 Storage Efficiency

In this sub-section, we will compare erasure codes it terms of Storage Efficiency. Storage efficiency is defined as:

Table VIII. Storage Efficiency of erasure codes tolerating 3 disk failure

| Erasure Codes | Total Number of Elements | Number of Data Elements |
|---|---|---|
| Star Code | $(p-1) \times (p+3)$ | $(p-1) \times p$ |
| Triple-Star Code | $(p-1) \times (p+2)$ | $(p-1) \times (p-1)$ |
| RS-Like Code | $(p-1) \times (p+3)$ | $(p-1) \times p$ |
| Blaum-Roth Code | $(p-1) \times p$ | $(p-1) \times (p-3)$ |
| Cauchy-RS Code | $L \times p$ | $L \times (p-3)$ |
| HDD1 Code | $(p-1) \times (p+1)$ | $(p-2) \times (p-1)$ |
| HDD2 Code | $(p-1) \times (p+2)$ | $(p-2) \times p$ |
| WEAVER Code | $2 \times p$ | $p$ |

Table IX. Storage Efficiency of erasure codes tolerating $\geq 4$ disk failure

| Erasure Codes | Total Number of Elements | Number of Data Elements |
|---|---|---|
| Rabin-Like Code | $(p-1) \times (p+r)$ | $(p-1) \times p$ |
| Blaum-Roth Code | $(p-1) \times p$ | $(p-1) \times (p-r)$ |
| Cauchy-RS Code | $L \times p$ | $L \times (p-r)$ |
| WEAVER Code | $2 \times p$ | $p$ |

$$\textit{Storage Efficiency} = \frac{\textit{Total \# of data elements}}{\textit{Total \# of elements (incl. parity)}} \tag{2.3}$$

The storage efficiency of HoVer codes is $\frac{d(n-1)}{(d+v)(d+h)}$, where $d$ is the number of data rows, $h$ is the number of horizontal parity disks, and $v$ is the number of vertical parity rows. Since the value of $d$ is not linear with value of $n$, we show the result in Figure 18.From Table VIII and Figure 18, we can see that WEAVER code has lowest storage efficiency ($50\%$). All MDS codes have the same storage efficiency and as the number of

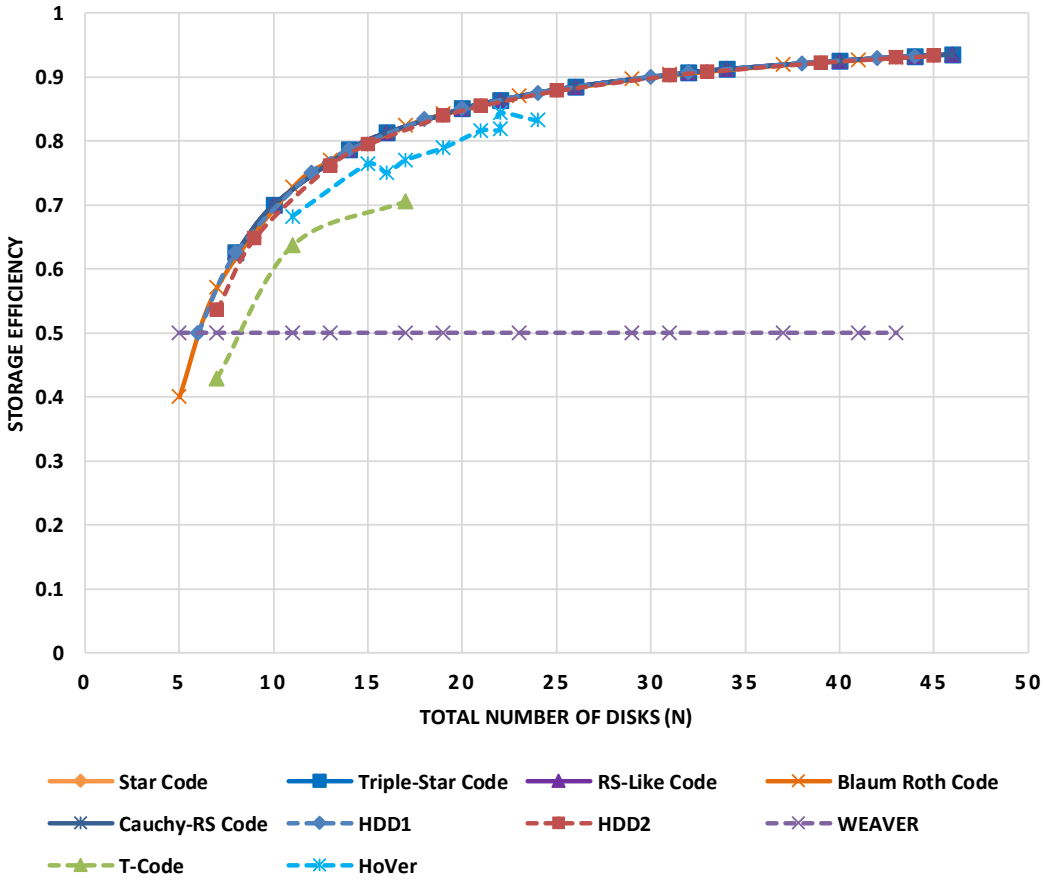



Fig. 18. Storage Efficiency of erasure codes tolerating 3 disk failure

disk increases, the storage efficiency also increases. Although T-code has a better storage efficiency than WEAVER, its storage efficiency is less than other Non-MDS codes.

In the case of 4 fault tolerant case also, WEAVER code is the least storage efficient code (see Figure 19 and Table IX). Since the storage efficiency of HoVer code is close to MDS codes, this code is also called Near-MDS code.

### 2.2.5 Rebuild/Reconstruction Efficiency

In this subsection, we evaluate the rebuild/reconstruction efficiency, which is total XOR operations required per failed symbol including both data and parity symbols. We

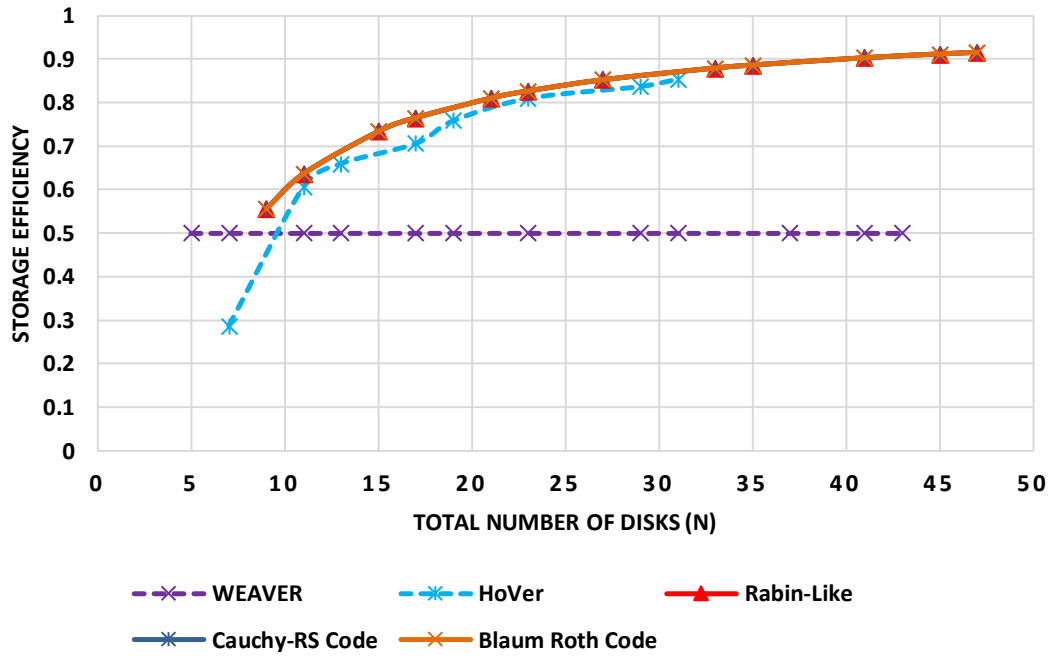


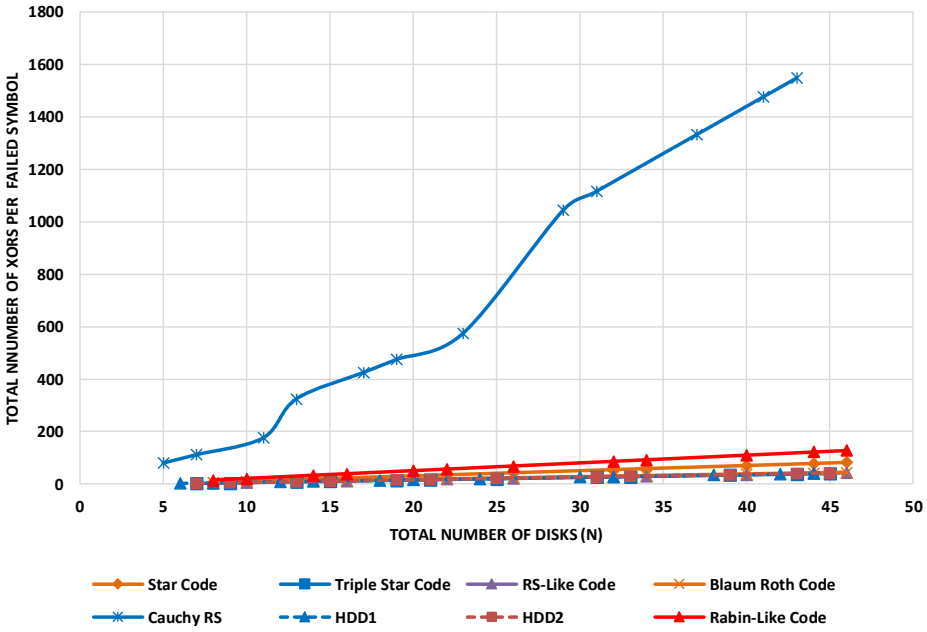

Fig. 19. Storage Efficiency of erasure codes tolerating 4 disk failure

evaluate cases of one, two, three and four disk failures.

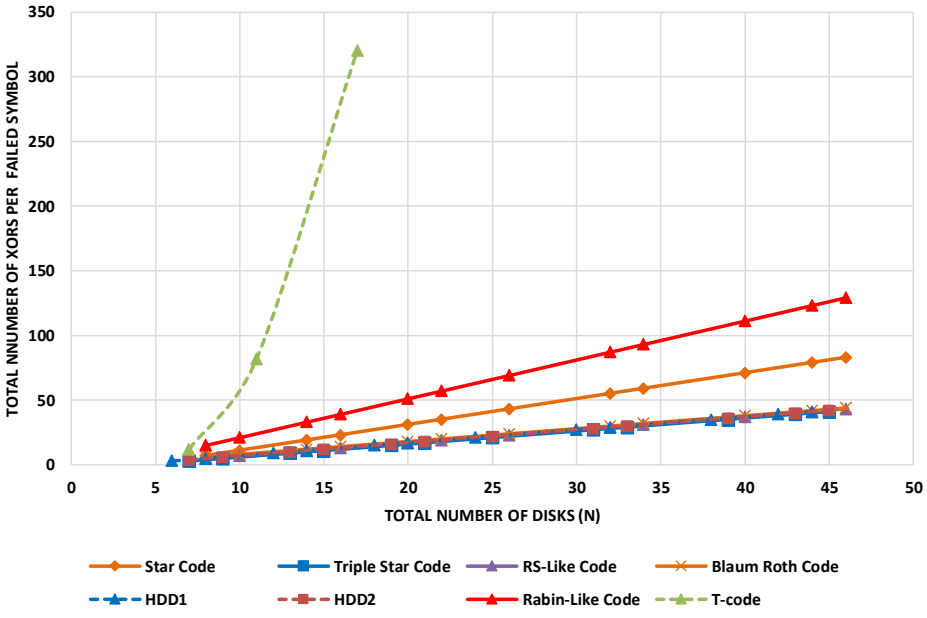
In the case of one parity disk failure, most of the codes have almost similar reconstruction efficiency and among these codes Triple-Star code has highest efficiency, but the difference with HDD2 and HDD1 code is not so significant (Figure 20(a)). Since Cauchy-RS code has worse reconstruction efficiency, we do not compare it with other codes except in Figure 21(c), where T-code has the worst reconstruction efficiency because of the instability of T-code decoding algorithm.

In Figure 20(a),20(b), 21(a), 21(b), 22(d) and 22(b), HDD1 and HDD2 codes have better performance than Rabin-Like Code and Blaum-Roth code because to recover failed symbols, we do not need to go through all the steps of the decoding algorithm in [71]. In other cases because of Gaussian Elimination and re-encoding of parity disks in the HDD1 and HDD2 decoding algorithm, these codes have lower reconstruction efficiency.

Although HoVer codes have lower reconstruction costs in most of the cases, due to




(a) One Disk Failure (Parity Disk)




(b) One Disk Failure (Data Disk)

Fig. 20. Rebuild/Reconstruction Efficiency for Erasure codes when 1 disk fails.

instability of HoVer codes and longer reconstruction chains, it needs to read nearly all data symbols in other cases. Thus, HoVer codes did not outperform Star code and Triple

Star code in 22(d).The reconstruction cost increases linearly with the increase in the total number of disks for all the codes except T-code, which has exponential increase and almost linear increase for Cauchy-RS code.

## 2.3 Observations

In this section, we list some of the important observations of our analysis and comparison.

- High encoding/decoding complexity in Blaum-Roth Code [11] makes it less efficient for storage systems with fewer disks. However, as the number of the disk increases, its performance increases as encoding/decoding operations require around 3 XOR operations per data symbol.

- Among MDS codes evaluated in this chapter, Cauchy-RS Code [12], has the least encoding/decoding/rebuild efficiency, and increases with an increase in the number of disks because the encoding/decoding complexity is roughly proportional to Galois parameter $L^2$ and to increase $n$, we also need to increase $L$ as $(n - r) \leq 2^L$.

- Triple-Star code [74] has the highest encoding/decoding efficiency among MDS codes evaluated in this chapter because it does not use *adjusters* (which are used in STAR-code) during encoding/decoding procedure. Reduction in the number of symbols to encode and decode reduces total XOR operations required for encoding/decoding, thus reducing encoding/decoding complexity.

- The decoding algorithm of T-code [70] is unstable because during decoding procedure, a parity symbol is randomly chosen and if only one information symbol among those contributing to that parity is erased, we proceed otherwise we select another parity symbol. Sometimes, this algorithm cannot correct all erasures and iterating

through all parity symbols randomly leads to a very low decoding efficiency for T-code.

- Although WEAVER codes [35] have a high fault tolerance, if the storage efficiency is of concern, it is efficient to choose other MDS codes as WEAVER codes can only provide a maximum storage efficiency of $50\%$.

- If encoding/decoding efficiency is of utmost importance, and the storage efficiency is of no concern, WEAVER code is suited because the total XOR operations required to encode/decode/rebuild a data symbol is not only fairly low but also constant even if the number of disks is increased in the system.

- There is a trade-off between storage efficiency and encoding/decoding complexity in all the evaluated erasure codes. For example, WEAVER code gains an improvement in encoding/decoding complexity by limiting the storage efficiency to a maximum of $50\%$, whereas MDS codes have optimal storage efficiency, but higher encoding/decoding complexity.

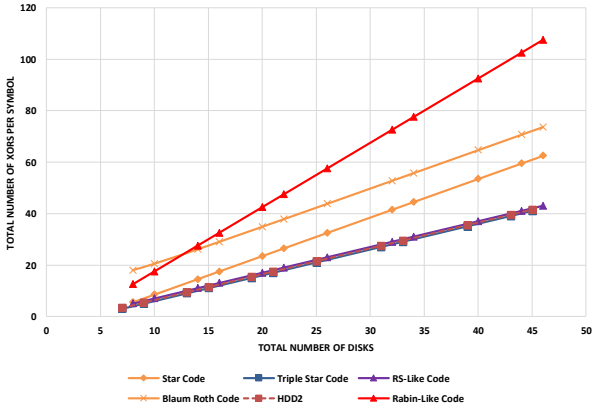
- If the number of disks in a system is fairly high ($\sim 100$), there is a minimum difference in storage efficiency among erasure codes except T-code [70] and WEAVER code[35].

- In most of the cases, HoVer codes [36] have better reconstruction costs. However, in cases of clustered failures, long reconstruction chains in these codes reduce overall reconstruction performance.

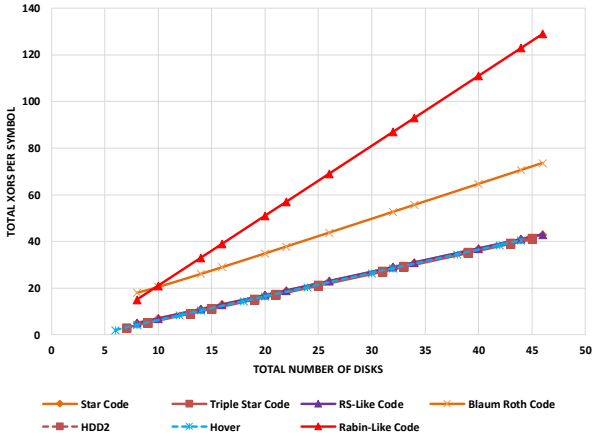
## 2.4 Summary

In this chapter we present a comprehensive analysis of popular erasure codes. These codes were classified as MDS and Non-MDS. We evaluated these codes based on encod-

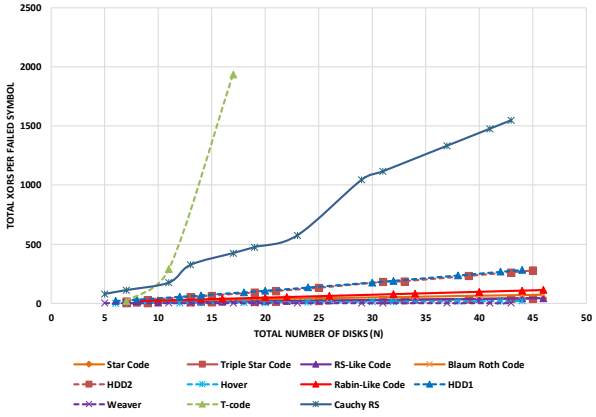
ing/decoding efficiency, storage efficiency and rebuild/reconstruction efficiency. We observed that there is a trade-off between the storage efficiency and performance of erasure codes. Codes such as HoVer and T-code have flexible parameters that give designers the ability to achieve the best configuration tailored to their needs.




(a) Two Disk Failure (Two Parity Disks)

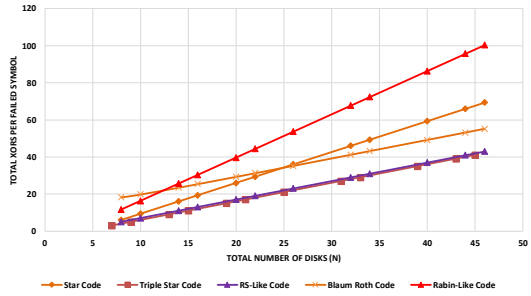



(b) Two Disk Failure (One Data Disk and One Parity Disk)

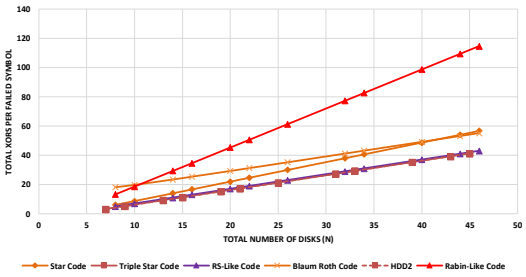



(c) Two Disk Failure (Two Data Disks)

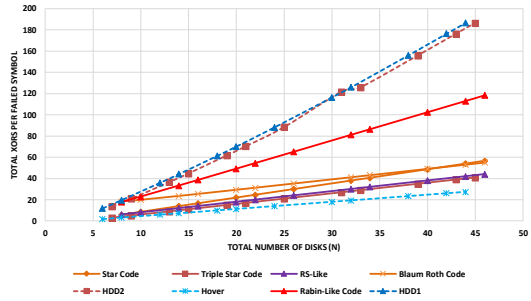
Fig. 21. Rebuild/Reconstruction Efficiency for Erasure codes when 2 disk fails.



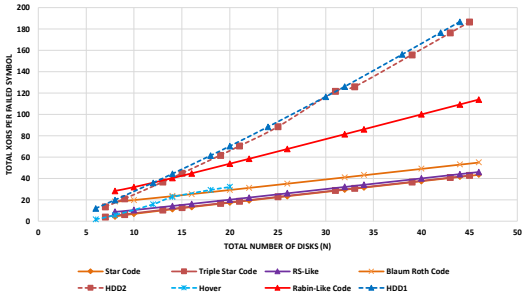

(a) Three Disk Failure (Three Parity Disks)




(b) Three Disk Failure (One Data Disk and Two Parity Disks)




(c) Three Disk Failure (Two Data Disks and One Parity Disk)




(d) Three Disk Failure (Three Data Disks)

Fig. 22. Rebuild/Reconstruction Efficiency for Erasure codes when 3 disk fails.

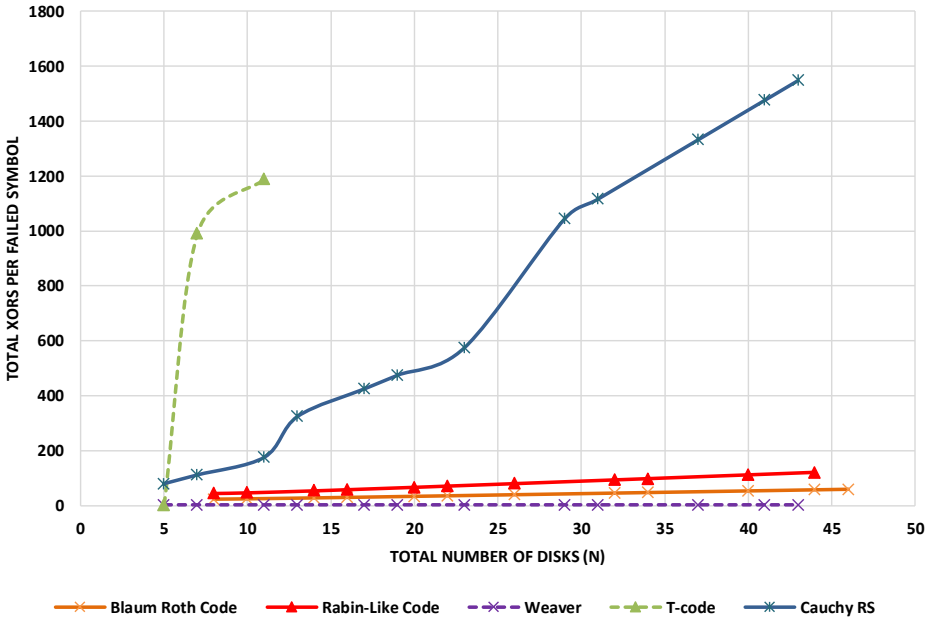



Fig. 23. Rebuild/Reconstruction Efficiency for Erasure codes when 4 data disks fail.

# CHAPTER 3

# HYBRID ERASURE CODED ECC SCHEME (EECC) FOR SOLID STATE DRIVES

In this chapter we present a hybrid erasure-coded ECC scheme (EECC), in which weak-ECC is employed to improve read performance and erasure code is used to bolster the reliability of SSDs. This design strategy targets read-intensive applications, such as web search applications, where read performance of the system is the main concern. BCH-code's decoding speed is primarily dependent upon the code word length and tolerable bit failures, which is the motivation behind this chapter. Current SSDs use BCH codes that have long code-word length equating to the page size. Using shorter code-word ECC, termed as weak-ECC in this chapter, we can speed up the decoding process. If we partition each page into smaller segments, with each segment being protected by shorter and weaker ECC, we can decode each segment independently. During a page read, this design strategy allows us to overlap the flash-to-controller data transfer of one segment with the decoding of another segment. Due to this pipelining effect, we dramatically reduce the page read latency.

In SSDs the raw page error rate is directly proportional to the program and erase cycles and thus at some point, the weak-ECC is not able to provide the desired level of reliability. Thus we propose the use of erasure codes in conjunction with weak-ECC to achieve, better read performance, much higher reliability, and increase the program and erase cycles the SSD can tolerate. For the recovery from erasures in erasure codes, we need to know the location of erasures/failures. In our design, weak-ECC serves two purposes. First, it can recover from most of the bit failures [52] and second it serves as an error locator

for erasure code. Erasure codes are used in segment level, where an HDD is used to store the parities/redundant-data associated with each segment level stripes. Direct use of erasure codes in SSD (without using the HDD) leads to increase in storage cost as SSDs are still more expensive than HDDs and storing the parities in SSD adds to the number of writes, consequently increasing the P/E cycles and thus reducing the effective lifetime of the SSD. The use of HDD also removes the parity update operation from the critical write path. We use a log-based write approach to storing parities in the HDD, so that the write performance of HDD does not become the bottleneck of the system. In our design, as soon as the data is updated in the SSD, the write completion is reported back and the parity update in HDD will be performed in the background. Moreover, most of the frequent writes in both HDD and SSD are served by the cache present in both HDD and SSD, hiding the overall number of writes and write latency seen by both HDD and SSD.

### 3.1 System Design

Motivated by the development of SSDs for use in high end storage systems, such as web servers, where the read latency is of critical concern along with the reliability of data, we come up with a hybrid erasure-coded scheme (EECC) architecture, which couples the erasure code with weak-ECC for improved read performance and reliability. The point to be noted is that, this work targets read-intensive applications. During the coupling of erasure code and ECC, we propose to use a high performance HDD to store the parities. Our design can be easily extended to use storage class memories, such as NVRAM, instead of HDD to store redundant data for even faster performance. While other designs focus on managing the wear-leveling or applying the RAID configuration at the page level to improve reliability, we focus on using erasure codes at the segment level to leverage the parallelism across multiple flash chips. We define reliability stripes in the flash translation layer (FTL) and the host/controller has direct access to the pages/segment in the device.

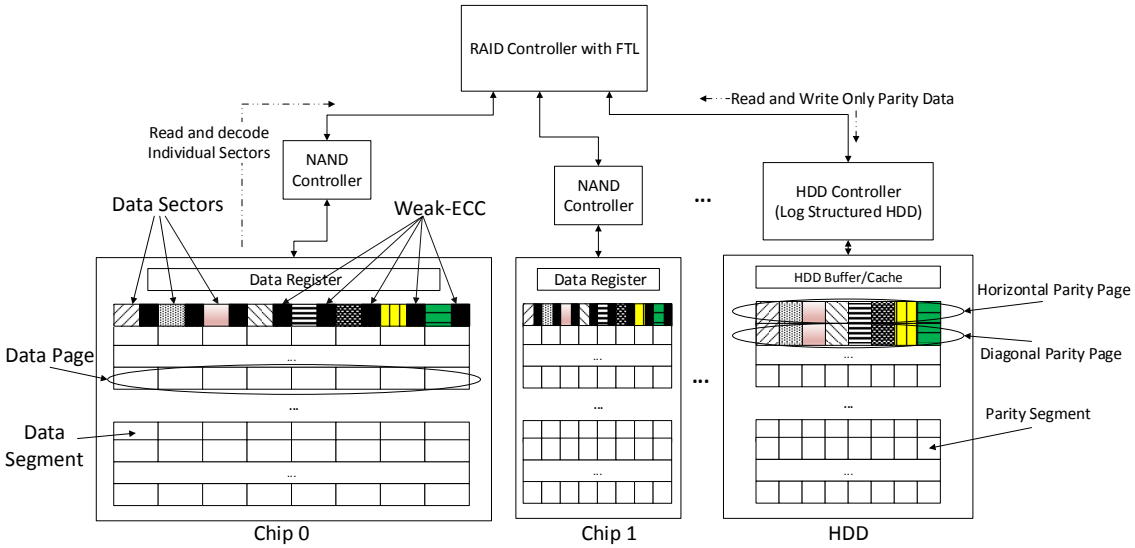



Fig. 24. EECC system architecture showing multiple chips (each page is broken down into 8 segments and each segment protected with shorter and weaker ECC) and the HDD (log structured and storing parities for stripes formed with only one segment from each page in different flash chips)

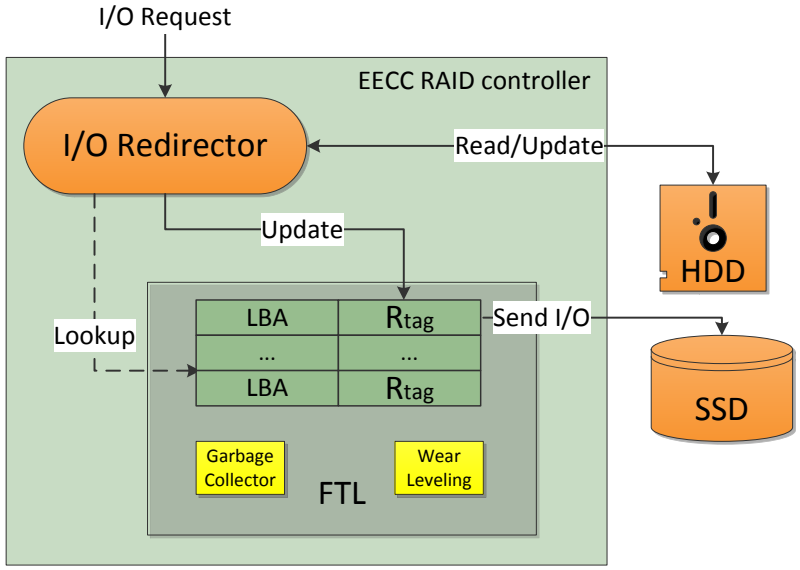



Fig. 25. EECC system architecture detailing the RAID controller

The naive or out-of-box use of erasure codes in the SSDs leads to the contention in the flash chips, due to "read-modify-write" operation of parity pages and frequent updates on these pages. We control the way of writing the parity to the hard-disk, in order to minimize the frequent parity updates. In this chapter, we achieve following goals:

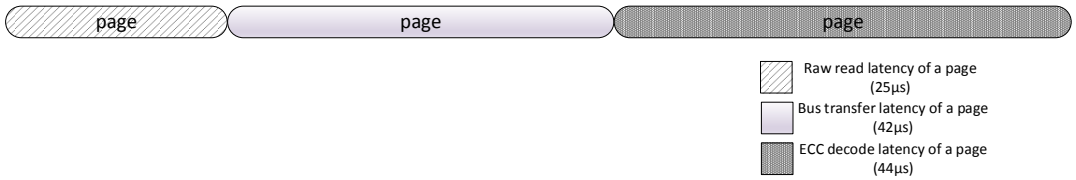
1. **Improve the read performance**: In systems like web-search, database analysis and search, data center virtualization, and web hosting, read latency is of main concern because we want to provide the results as soon as possible to the application user. Although current SSDs have faster read performance than traditional hard-disks, we further improve the read performance of SSDs with the use of pipeling during page reads.

2. **Increase the reliability**: Current SSDs rely on ECC to tolerate from bit errors. Due to the use of complex ECC schemes to improve the reliability, these solutions have slower read performance. In addition, these schemes do not tolerate burst, device and chip-level errors. Our work fulfills this objective with the use of erasure codes. We also use a log structured HDD to store parities, so that SSD performance is not aggravated by frequent parity updates.

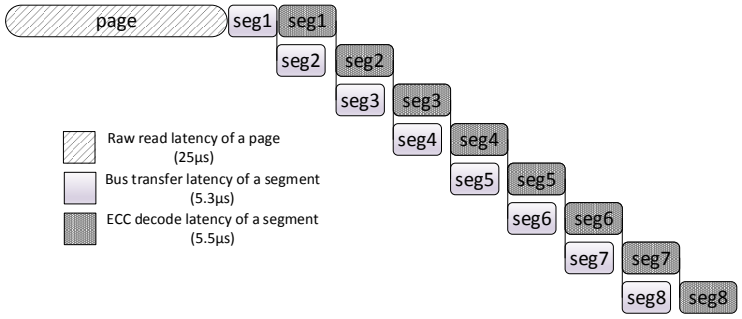
3. **Reduce overhead caused by parity updates**: In our work, we achieve this goal by adding a high performance HDD to store parity information, so that frequent writes incurred due to parity updates do not aggravate SSD performance. The parities are written in a log-structured manner in the HDD.

### 3.1.1 System Architecture

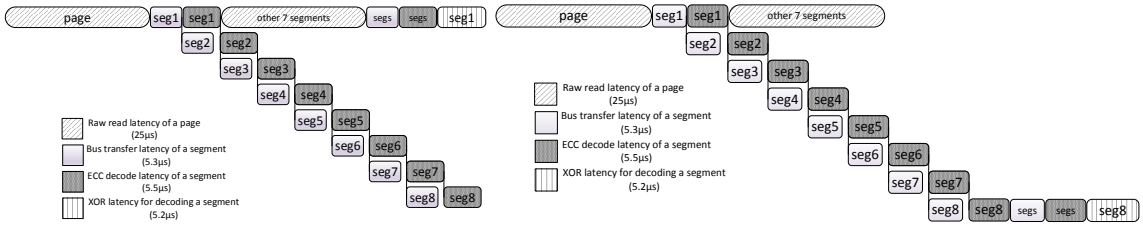
In this sub-section, we describe in detail our proposed design and then discuss the potential speedup due to the use of weak-ECC coupled with erasure code. Figure 24 illustrates the basic architecture of the proposed EECC design.




(a) Read latency in regular SSDs, where we sense the whole page from the chip to the data register, then transfer through the bus, and decode in the ECC engine




(b) Read latency in EECC in regular cases, where we sense the whole page from the chip to the data register, then overlap individual page segment/sector bus transfer and decode time i.e. pipeline 8 segment's transfer and decode time (thus low read latency than regular SSD)




(c) Two cases of read latency (seg1 failure or seg8 failure) in EECC during the requested segment failure, where we first sense the page from the chip to the data register, then transfer the page to ECC engine and sense other segments in the reliability stripes, decode it and if error is found(seg1 or seg8), start the recovery procedure for that segment (transfer reliability segments to the ECC engine, decode them and recover the segment in RAID controller)

Fig. 26. Read latency in regular SSDs employing $(33408, 32768, 40)$ BCH-code and EECC employing $(4226, 4096, 13)$ BCH-code

The EECC design consists of a RAID controller implemented above the SSD and HDD. The Chips 0 and 1 are the SSD flash chips and our design can support any number of flash chips and uses a single HDD for parity storage. The RAID controller is capable of

controlling $n$ flash chips and one HDD simultaneously. Since we move the flash translation layer (FTL) off the SSD and onto the RAID controller, the controller is capable of managing each page and segments individually. The FTL can also be moved to the host and the host can optimize the writes in a way to maximize the write performance. [18] demonstrated that we can move the FTL off the flash chips and implement the FTL into a separate controller or the host. Thus, in this chapter, we do not discuss how we can move the FTL off the SSD and into the RAID controller. As each flash chips have dedicated NAND controller, the ECC decoding and encoding operation across individual chips is done simultaneously.

Figure 25 shows in detail the RAID controller for EECC architecture. As mentioned above, the FTL is moved out of the SSD into the RAID controller. The additional changes for the EECC implementation is the addition of the $R_{tag}$ to the LBA mapping table in the flash translation layer. The $R_{tag}$ is used to associate each segment to the RAID stripe. The I/O redirector is in charge of sending the I/O requests to the SSD and HDD. For every request, it identifies the read and write operations and then looks up the extra segments that are to be read or updated in the SSD and HDD for serving the request. Our current implementation utilizes the tree-based binary structure to handle the mapping of LBA and $R_{tag}$. The time complexity of the lookup operation is thus $O(logk)$. Concerning the memory overhead, for every block in the SSD, we need 9 bits in the $R_{tag}$ field, which is an acceptable requirement for the RAID controller.

Unlike regular SSDs, we change the way read requests are served. In regular SSDs, the read requests are served in page size, but we segment each page into segments and then send each segment individually to the RAID controller, and then the controller serves the data to the host. Figure 26 shows the potential speedup we can achieve from changing the way the read requests are served. In regular SSDs, each page is protected by a single and complex ECC. Thus, during the page reads, we first need to sense the page from cell to buffer, i.e., the raw page read latency is the time required to transfer the page to the data

register in the flash chip. Then, we need to transfer the page via bus to the ECC engine to decode and if any errors are detected, correct them. In our design, each small segment is encoded with shorter and weaker ECC. As each segment now has independent ECC, the bus transfer time and ECC decoding time of each segment can be overlapped.

### 3.1.2 Erasure Coding across Flash Chips

As mentioned in Section 1.1, the reliability of SSD is reducing due to the use of MLC flash devices. Our proposed design couples the faster read performance of weak-ECC with the high reliability of erasure codes, and is able to recover even from double chip failures. The coupling of an erasure code along with weak-ECC serves two purposes. First, the weak-ECC tolerates from most of the common bit errors, thus we need-not go through a costly data recovery procedure of erasure code. In erasure codes for data recovery, we must know the location of erasures before we can begin the data recovery procedure and weak-ECC serves also as an error locator, because it reports segment error for the occurrence of more bits errors than it can tolerate.

For erasure coding across flash chips, we form a reliability stripe out of $n$ segments in the SSD, with $n$ flash chips, and 2 segments in HDD, where each flash chips contribute single segment for data and 2 segments from HDD are used for horizontal and diagonal parity. The reliability stripes formed are not hard-coded i.e. the stripe formation is logical in the RAID controller. When a data is to be updated, we just update the logical pointer i.e. $R_{tag}$, which is forming the reliability stripe, in the FTL to point to the updated data. The reason behind this strategy is the lack of "in-place update" mechanism in SSD. For evaluation purposes, in this work we used RDP-code [22] as the erasure code. With the minimal change in the RAID controller for logical stripe formation and no change in SSD or HDD, the system can be easily extended to use any erasure code.

In EECC, a page is divided into 8 segments and only one segment from each page

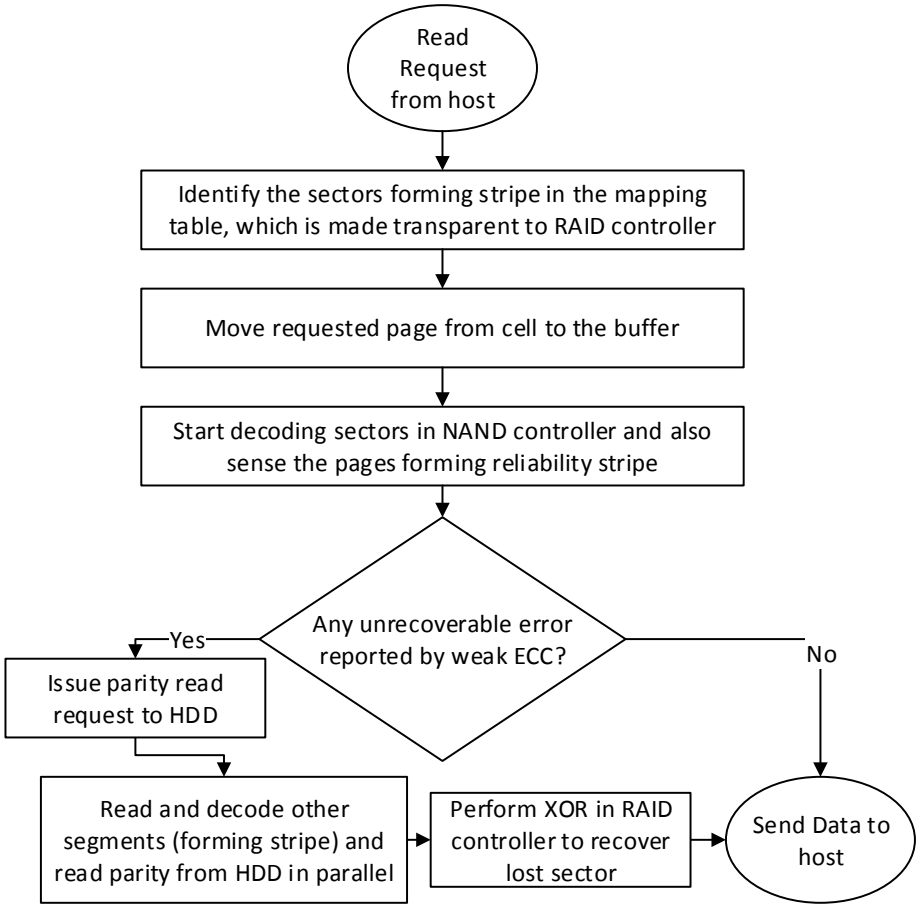



Fig. 27. Flow diagram for page read operations in EECC

in different flash chips contribute to the stripe formation. Figure 24 illustrates a stripe formation, where same pattern segments form a stripe, i.e., one segment from one flash page of a flash chip is used as a data symbol for stripe formation. Thus, the whole system is equivalent to a group of 8 RAID-arrays employing erasure code. The horizontal and diagonal parities are stored as segments of different pages in the HDD, and these pages are organized in a log-structured manner. After the HDD buffer becomes full, the parities are committed to the disk. As we maintain the log structure for parities and write them in a sequential manner, the performance of SSD is not limited by the read/write response time of the HDD. Since the parties are updated in HDD in the background, the parity update

operation is removed from the critical write path.

Figure 27 shows how a read request is served in EECC. Since we employ weak-ECC to tolerate common bit failures, most of the raw-bit-errors are recovered by the ECC. When the program and erase cycles increase, the bit error rate increases in the SSD and the ECC is unable to recover from bit errors. It reports failure to the RAID controller and the RAID controller begins the recovery procedure for the failed segment. Figure 26(c) shows the pipeling and overlapping of the decoding and recovery procedure during a single segment failure, when a page read request is being served. For example, let us consider that a page is requested from chip 0, and during the read of the first segment, the weak-ECC reports failure. Then we need to sense other segments from the same logical stripe, and because of parallelism in SSD, 7 segments can be sensed at the same time across 7 flash chips. They can also be transferred and decoded by the ECC engine within the same time-window. At the same time, the horizontal and diagonal parities can be transferred from HDD to RAID controller without affecting the SSD. Thus, all processes can be overlapped. Since all data necessary for recovery is now present in the RAID controller, the controller can do XOR operations for segment recovery and this XOR operation can also be overlapped with bus transfer time and decoding time of other requested segments.

During a write/update request, first the RAID controller identifies the logical stripe, with the help of $R_{tag}$, the new data corresponds to. Then it updates the $R_{tag}$ field for the new LBA, and at the same time issues read request to the SSD and HDD for parity computation. The read requests are only for the old parity and the old data (if it is to be updated). If the data is new i.e. write operation (not update), then we need not send the read request to the SSD, because we can just recompute new parity by XORing the recently written data with the old parity. Now the newly written data and the newly computed parity are sent to the SSD and HDD respectively. Thus for update operation of a page, we have an overhead of single page read from SSD and two page reads from the HDD. Since we target read-

intensive applications, the re-computation of the parity data and writes to the HDD can be done in the background (outside of the critical write path). For update operations, the SSD needs to write data to the new page, instead of overwriting the same page. However, this does not change the logical stripe formation of erasure code. We simply change the mapping of the logical stripe in FTL i.e. update the $R_{tag}$ to point to the new page.

Our design sacrifices some write latency for improved read performance and increased reliability. We believe that increase in write latency is overshadowed by the significant reduction of read latency because this design in targeted for read intensive applications, where the reads constitute more than $80\%$ of the total workload. We discuss in more detail about the performance gain we achieve and the overhead for the write intensive applications in the following sections.

## 3.2 Evaluation

This section evaluates the efficiency of our proposed EECC design. We have implemented and evaluated our EECC design based on a series of comprehensive trace-driven simulation experiments on the modified Disksim simulator with SSD add-on from Microsoft [8]. In this section, we show some of the analysis and experimental results for comparing EECC with regular SSDs. We evaluate the average response time and the cumulative distribution function (CDF) of the workload response time along with the reliability in terms of code-word error rate.

### 3.2.1 Performance

For the performance evaluation, we show the potential gain of our work through a theoretical analysis of read response time and the simulation experiments evaluate the average response time and read response time. For the evaluation of write response time along with read response time, we also plot the CDF of workload responses.

Table X. Operational latency for EECC and regular SSD

| | BCH Decoding ($\mu s$) | Bus Transfer ($\mu s$) | Cell to Buffer($\mu s$) |
|---|---|---|---|
| Regular ECC | 43.91 | 41.76 | 25 |
| EECC | 5.57 | 5.33 | 25 |

Figure 28 shows the read latency for a range of fault tolerance in the ECC implemented. The erasure code implemented in EECC is RDP-code [22] and the ECC is a (4226, 4096, 13) BCH code, whereas the baseline system is the SSD armed with (33408, 32768, 40) BCH code for fault tolerance. The fault tolerant bits for regular SSD is per page, whereas for weak-ECC and EECC, they are per sector. The system termed as weak-ECC is a system which is not coupled with erasure code. We can see that fault tolerance and read latency are directly proportional and at after some point of fault tolerance, the read latency of regular ECC is better than weak-ECC and EECC. This is because of the weak-ECC and EECC having ECC in sector level, which increases the sector size. The increase in effective page size (including data and ECC bits) is more in case of weak-ECC and EECC than regular ECC, thus making the overhead larger for bus transfer and decoding of ECC. Table X lists the main relevant operational latencies. Since small size segments rather than whole pages are read and decoded, EECC has lower BCH decoding and Bus transfer time than regular SSD. Without loss of generality, in Figure 28 we assumed that the faulty segment is the first segment. Our simulation experiments do not make this assumption, because a faulty sector can be any segment in the pipeline. We observed that after $t = 19$, where $t$ is the fault tolerant bits in the ECC, the time required to read a page without sector failure and with sector failure is same. The reason behind this is the time required to transfer other remaining 7 sectors is greater than the recovery of the failed sector.

In our simulation experiments, we simulate an 8-chip 60GB SSD with a 3.2Gbps throughput [69], a RAID controller operating at 800Mhz [43] and an HDD with 3.0 Gbps

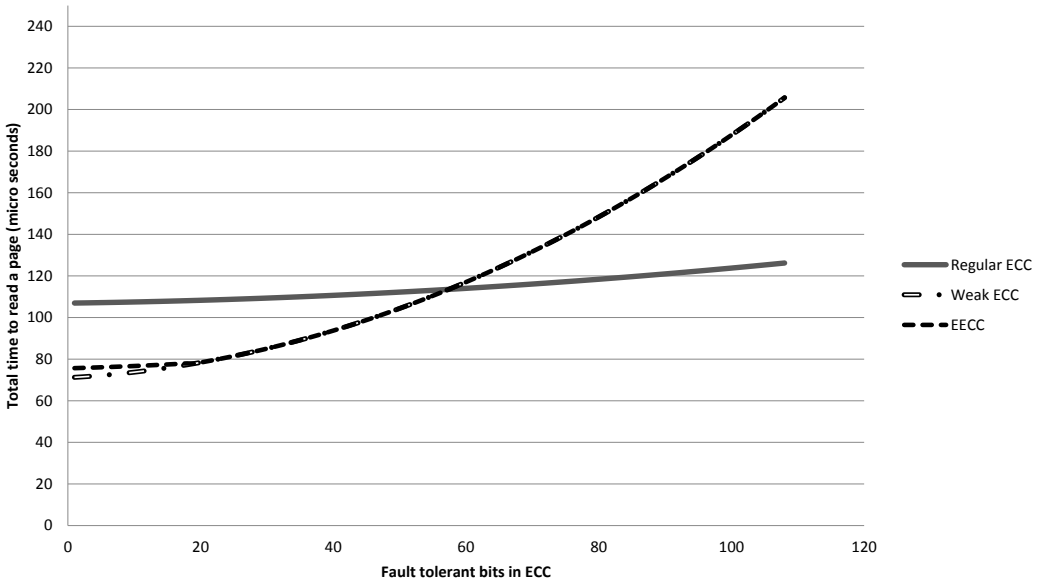



Fig. 28. Read latency for varying fault tolerance in ECC

Table XI. Trace Characteristics and Baseline Average Response Time

| | F1 | F2 | Websearch | Synth. 1 | Synth. 2 |
|---|---|---|---|---|---|
| Read | 28.47% | 86.02% | 100% | 95.12% | 98.03% |
| Write | 71.53% | 13.98% | 0% | 4.88% | 1.97% |
| Average Response Time $\mu s$ | 0.523 | 0.176 | 0.118 | 0.136 | 0.124 |

throughput for sequential reads. We replayed few real-world disk I/O traces, whose characteristics are detailed in Table X, in our simulation experiments. *Financial1*, *Financial2* (F1, F2) [68] were obtained from OLTP applications running at two large financial institutions and *WebSearch2* (WebSearch) [68] was from popular search engine. *Synthetic1* and *Synthetic2* traces were generated within the Disksim simulator [13]. Table XI shows read and write characteristics of traces that were subject to evaluation. It also shows the average response time of the baseline system i.e. the SSD with regular ECC.

As discussed in earlier sections, SSD read requests are prone to errors and for simplic-

Table XII. Corruption related statistics for different segment error rates

| | F1 | F2 | Websearch | Synthetic1 | Synthetic2 |
|---|---|---|---|---|---|
| SER = $10^{-3}$ | 1 | 11 | 59 | 2 | 2 |
| SER = $10^{-4}$ | 12 | 103 | 599 | 27 | 27 |
| SER = $10^{-5}$ | 114 | 1037 | 5789 | 373 | 380 |

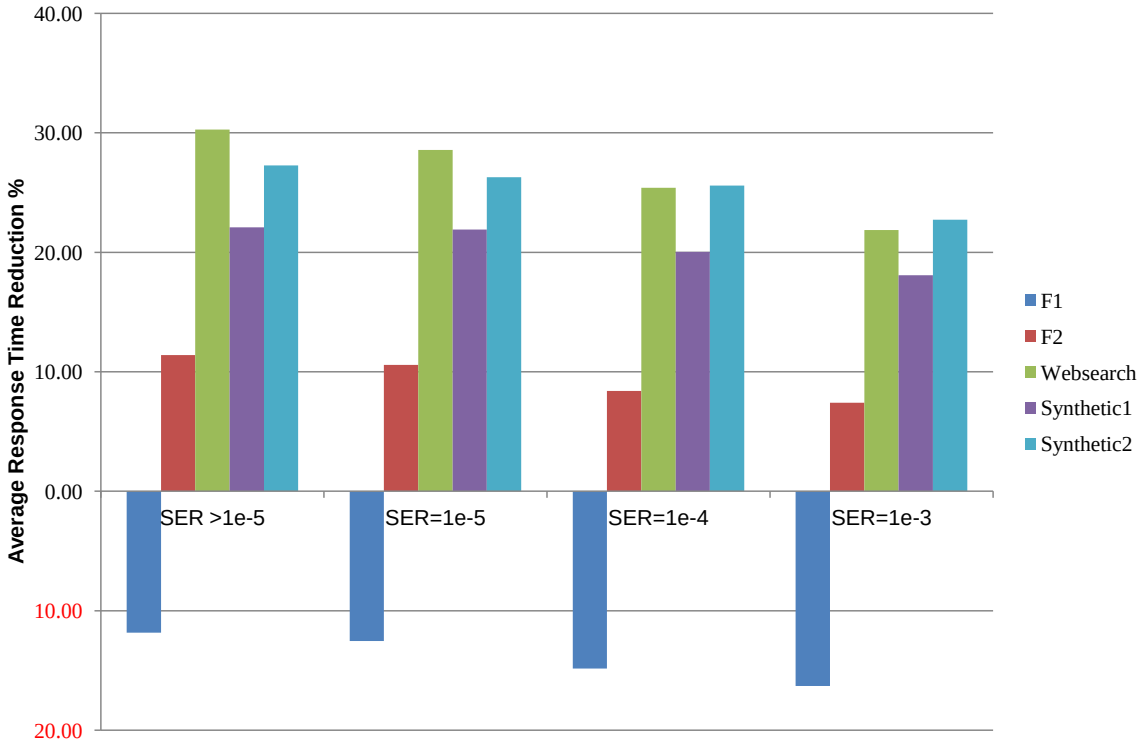
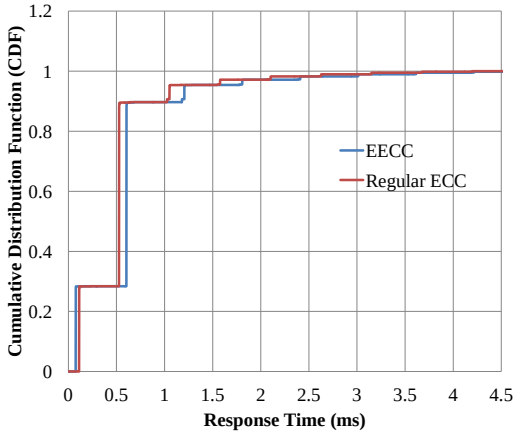
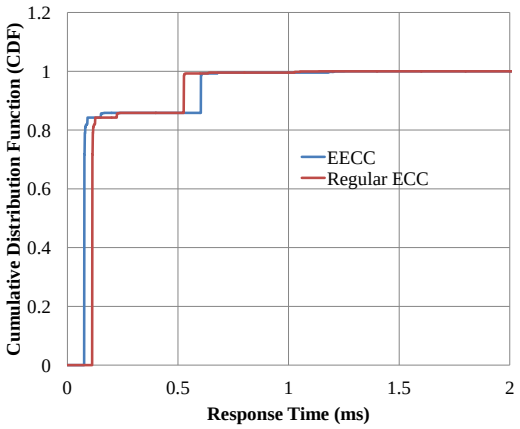



Fig. 29. Workloads average response time reduction, normalized to baseline, for varying segment error rates

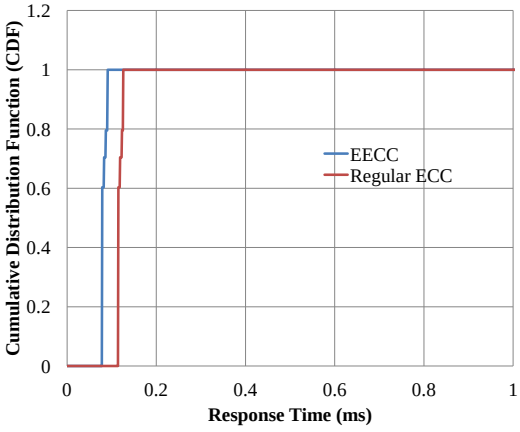
ity but without loss of generality, we introduce errors to the flash pages according to specific segment error rates. Table XII shows the errors introduced in flash pages and for generality, we the occurrence of faulty sectors i.e. can be segment 1 or segment 8 or any other segment in between. While writes are inherently directed at the FTL layer and thus bypass faulty pages, we assume faulty pages only affect reads. Figure 29 shows the average response




(a) F1 CDF

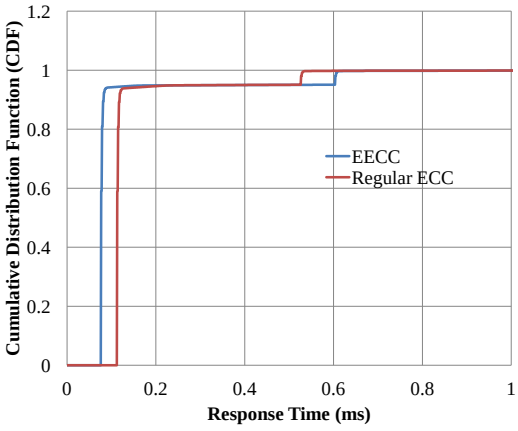



(b) F2 CDF




s

(c) Websearch2 CDF




(d) Synthetic1 CDF

Fig. 30. Workload's response time cumulative distribution function (CDF) comparison

time reduction for varying segment error rates. We see that for read intensive workloads, the average response time was improved up to $30.28\%$. With more faulty pages, the average response time gain decreases due to the need to recover the faulty pages with costly erasure code. Figure 30shows the response time for the workloads. It can be seen that a certain percentage of responses have smaller response time for EECC and these percentage of responses are equal to the read percentages of the workloads. To further demonstrate the improvement in read response time, we compared the average read response time in

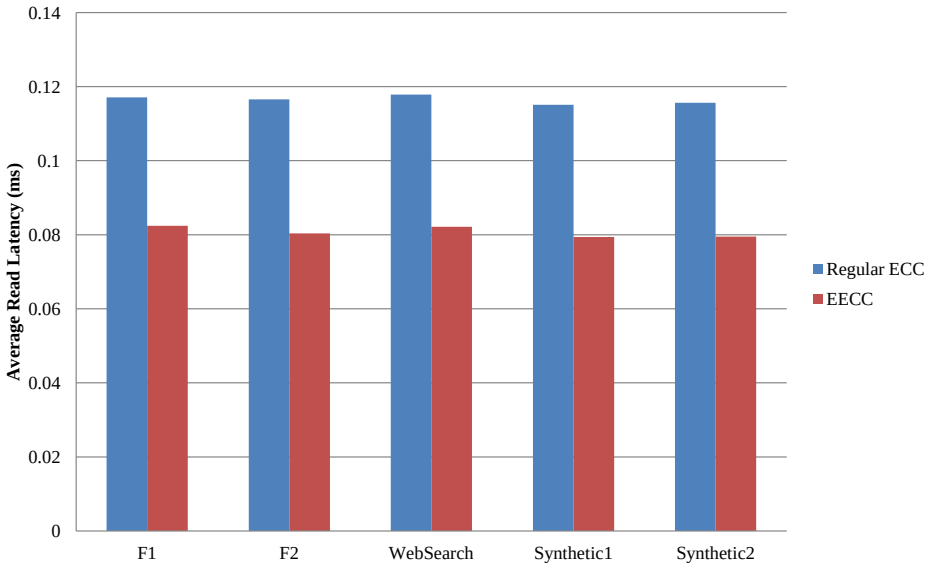



Fig. 31. Read response time for different real-world traces

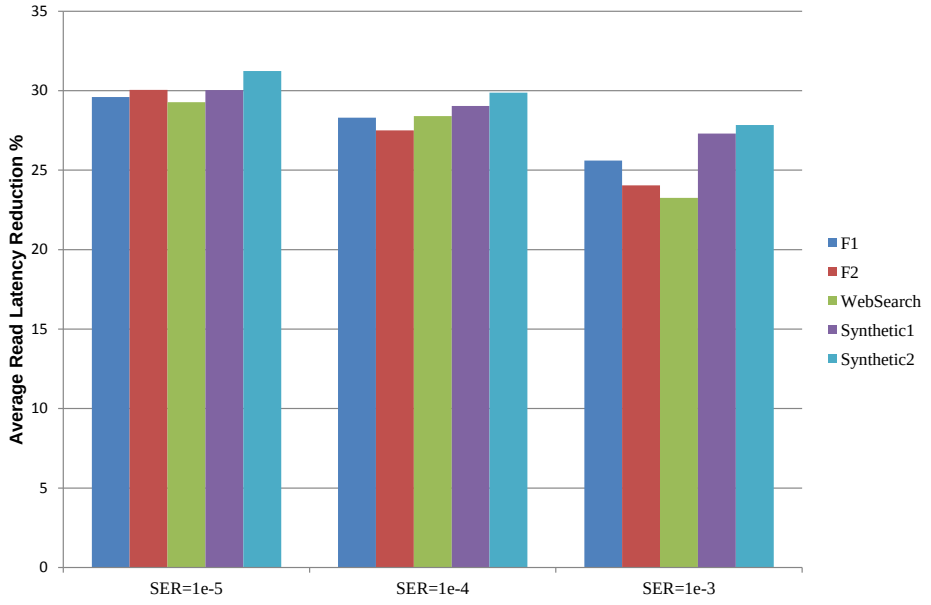



Fig. 32. Workload average read response time reduction for different real-world traces

Figure 31 for the case when there were no faults or all the faults were recovered by the weak ECC, where we observed that EECC reduces the read latency by up to 31.23% and Figure 32 specifically shows the read response improvements normalized to baseline for

varying faulty conditions.

We observed that our proposed design outperforms regular ECC in all cases of read-intensive workloads the amount gain we can achieve depends on the percentage of reads and writes the workload has. For write heavy workloads such as F1, there is reduction in performance and this overhead is due to increased write latency. The increase in write latency is because of parity update for every single write, causing a read-request for the old data along with the old parity. As the writes can be absorbed through the use of various write-cache techniques to reduce the number of writes seen by the SSD or HDD [77], the increase in write latency is not much of a concern. We target read-intensive applications and the writes constitute only small portion of the large workload, thus the small increase in write latency is overshadowed by the large reduction in write latency. From our evaluations, we conclude that EECC has a significant performance gain in read intensive applications, at the sacrifice of some write performance.

### 3.2.2 Reliability

In our evaluation setup, we implemented RDP code with $p = 11$ and shortened it to accommodate $8$ flash chips, because we do not want a reliability stripe size more than the number of flash chips. If the stripe size is more than number of flash chips, we cannot recover from faulty segments from remaining segments in parallel, because some flash chips will have more than one segment forming the stripe and this leads to increase in read latency. The estimated uncorrectable page error rate (UPER) for the system was derived from the raw bit error rate (RBER) by the use of cumulative Binomial Distribution. The cumulative binomial distribution is defined as

$$F(k; n, p) = \sum_{i=k+1}^{n} \binom{n}{i} p^i (1-p)^i \tag{3.1}$$

If a segment of length $S$ is encoded with $T$ fault tolerant bits, then the segment error

rate (USER) is given by $USER(T) = F(T; S, RBER)$. Since our system implements RDP code, which is double fault tolerant code, and the number of chips that can fail is 8, the new page error rate is given by

$$UPER = \frac{F(2; 8, USER)}{8} \tag{3.2}$$

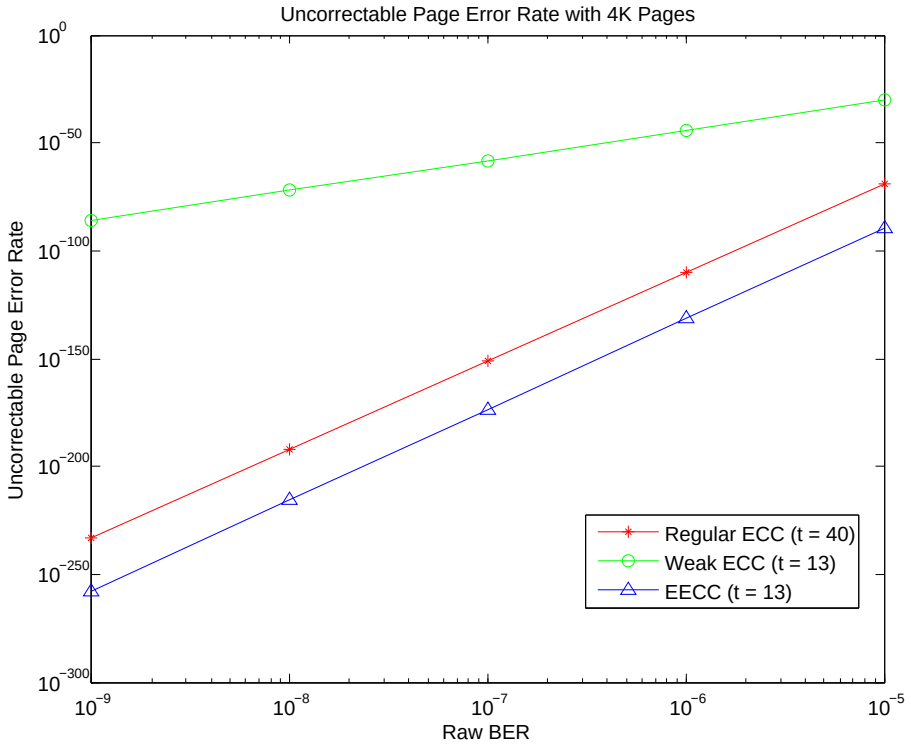



Fig. 33. Estimated uncorrectable page error rate

The reliability of a system is illustrated by the UPER the system can tolerate and it is shown for EECC, weak-ECC and regular ECC in figure 33. The range of RBER was set from $10^{-9}$ to $10^{-5}$ for UBER computation. Although, weak-ECC has better read-performance than regular ECC, there is a huge gap between the reliability of regular ECC and weak-ECC. Since the EECC can recover even from double chip failure, it has better

Table XIII. Number of program/erase cycles that can guarantee $10^{-15}$ UPER

| | Acceptable RBER | P/E Cycles |
|---|---|---|
| Regular ECC | $2.6 \times 10^{-4}$ | $8.1 \times 10^{3}$ |
| EECC | $8 \times 10^{-4}$ | $1.3 \times 10^{4}$ |

reliability along with the reduction in the uncorrectable page error rate by a factor of $3.2 \times 10^{23}$ in comparison to regular ECC. As the raw bit error rate is directly related to program and erase cycle of an SSD, increase in tolerable RBER reflects increase in P/E cycles of the SSD. Table XIII demonstrates that EECC scheme can improve the lifetime of an SSD by $60\%$. The table XIII, was constructed by extrapolating a graph in [14], which plots the raw bit error rate for all errors at different number of P/E cycles.

### 3.3 Summary

In this chapter, motivated by the inherent parallelism of SSD architecture along with a gap in decoding latency between regular ECC and weak-ECC, we proposed a new scheme called EECC which couples weak-ECC with erasure code to improve the read latency and also improve reliability of the whole system. The key here is, the weak-ECC is able to recover from most of the common bit-errors and as the SSD wears out, the weak-ECC will not be able to recover from bit errors and the erasure code comes into play to recover from segment errors. We also proposed the use of high performance HDD to store parities in a log-structured manner for the alleviation of the SSD from potential write overhead that might incur, if redundant data are stored in the SSD. We also performed extensive simulations and evaluated the read and write response time along with the uncorrectable page error rate (UPER). Our scheme was found to reduce the average read latency by up to $31.23\%$, average response time by up to $30.28\%$ and the UPER by $3.2 \times 10^{23}$ with some write latency sacrifice.

# CHAPTER 4

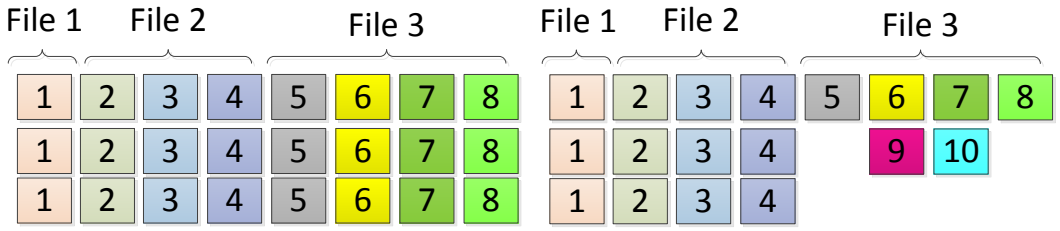
# FINGER: A NOVEL ERASURE CODING SCHEME USING FINE GRANULARITY BLOCKS TO IMPROVE HADOOP WRITE AND UPDATE PERFORMANCE

To improve the interoperability among various applications, the distributed file system needs to support basic operations like read, write and update operations. While read operation doesn't incur overhead during normal operations, for write and update parity needs to be re-computed and it requires reading other data blocks. One potential solution to reduce the amount of data being read during parity re-computation is to reduce the default block size. If the block size is small and we are updating the same data as before, more blocks are being re-written. This leads to a higher probability that all the blocks in the encoding stripe are being updated. But a smaller block size causes a significant increase in the metadata size of the Namenode in HDFS. Since the metadata information is stored in the Namenode's memory, an increase in Namenode metadata size becomes a bottleneck for scalability. Thus, the key motivation of this work is to perform encoding on large blocks such that updating any block in the erasure coded stripe incurs no extra read I/O associated with erasure coding and file recreation is eliminated, all while keeping the same metadata size as an HDFS-RAID instance utilizing a large block size.

We observe that limiting the update sizes to be a factor of the block size in HDFS can easily eliminate extra disk I/O during updates operations. To that end, we propose **FIN**e **G**rained **ER**asure coding scheme (**FINGER**), which is a new block-layout algorithm for erasure-coded Hadoop that enables efficient write and update operations.

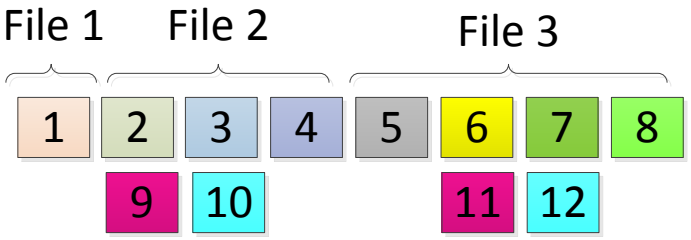
### 4.1 Motivating Example

In this section, we explain via a motivating example why the default erasure coding policy of HDFS hurts the write and update performance. We also provide intuitions on how to improve the write and update throughput. Let's review the replication and erasure coding policy in HDFS and HDFS-RAID respectively. By default, HDFS performs 3-way replication for all data blocks and stores the information of these blocks in the Namenode's memory as *metadata*.




(a) HDFS 3-Way Replication.

(b) HDFS-RAID File-level Encoding with RS(6,4) Code.




(c) HDFS-RAID Directory-level Encoding with RS(6,4) Code.

Fig. 34. Replication vs Erasure Coding for 3 files inside a directory.

Figure 34 illustrates an example, where an HDFS file directory contains 3 files. Let the block size be 128 MB. File 1, File 2 and File 3 are of 128 MB, 384 MB and 512 MB sizes respectively. In Figure 34(a) HDFS uses replication for data redundancy and it needs to maintain 3 copies of each block. In this case, HDFS requires a storage space of 3072 MB, i.e., a storage overhead of 200%.

HDFS-RAID performs erasure coding across blocks and it can be done in one of two ways: file-level encoding and directory-level encoding. In file-level encoding, blocks from the same file are used to compute parity while directory-level encoding computes parity by taking into account multiple blocks from different files inside the same directory. Figure 34(b) and 34(c) use Reed-Solomon(6,4) code for computing parities. RS(6,4) code takes 4 data blocks and generates 2 parity blocks for fault-tolerance against a max of 2 block unavailability. Since File 1 has only 1 block, it cannot be encoded to compute parity using RS(6,4) code because it doesn't have the sufficient number of blocks necessary to perform encoding. This case also applies to File 2. Thus both File 1 and File 2 needs to be triplicated to guarantee data availability, while File 3 can be encoded using RS(6,4) to compute 2 parity blocks. Thus, the storage overhead reduces to $125\%$ in Figure 34(b). In directory-level encoding (Figure 34(c)), erasure coding across multiple files reduces the storage overhead to $50\%$ because all 8 blocks can be erasure coded to calculate 4 parity blocks. So we can conclude that from storage overhead and the write-traffic perspective the directory-level encoding is optimal.

Now we discuss from the application's perspective if directory level encoding is optimal or not. Let us assume that File 1 is being updated or overwritten with new content. In replication, we only need to re-write new block content and replicate the block, i.e., it incurs an extra disk write I/O of 2 blocks beside the original block write I/O. In file-level encoding, since File 1 is still replicated, it incurs the same overhead as the replication. The directory-level encoding suffers extra disk I/O and computation overhead because changing the contents of block 1 invalidates the parities (blocks 9 and 10). So, we need to re-encode the stripe (blocks 1, 2, 3 and 4) to calculate new parities and update the old parity. This re-encoding of the stripe causes 3 extra block-read traffic and 2 block-write traffic, besides the original single block-write traffic. From this aspect, directory-level encoding performs worse because of extra computation and disk I/O.

From the example, we observed that there is a trade-off between the update performance and storage-savings/write-traffic. To gain advantage in one of them, we need to sacrifice another. Let us revisit the example, and suppose that we reduce the block size such that File 1 has exactly 4 blocks. In this case, file-level encoding works best, because updating that file changes the contents of the whole stripe and we need to perform only encoding and write new data and parities. If block size was 32 MB, we would incur an extra write overhead of 64 MB in contrast to 256 MB, i.e., 75% reduction in write/update traffic.

## 4.2 System Design

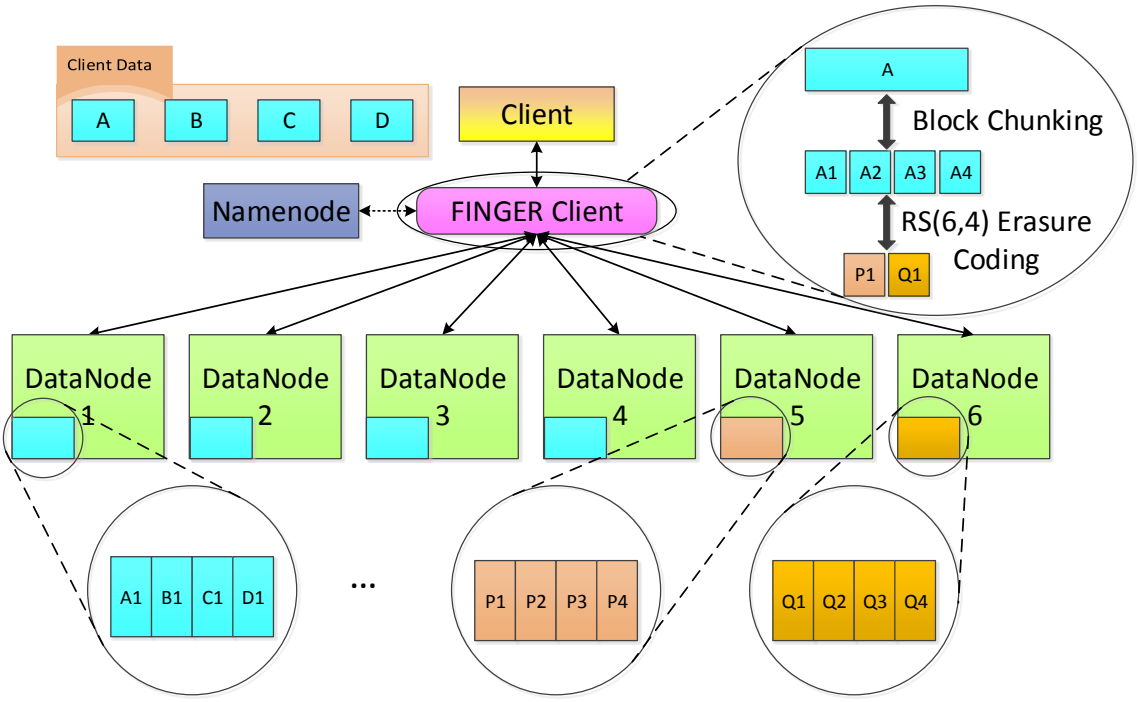



Fig. 35. FINGER System Architecture detailing Block-chunking, RS(6,4) erasure coding and final block-layout in the Datanodes. The smaller chunks from A, B, C, and D, i.e., A1, B1, C1, and D1 are appended together to create a larger block in Datanode 1. Similarly, parities from multiple blocks, i.e., P1 from A, P2 from B, P3 from C, and P4 from D are appended together to form a larger parity block.

Motivated by the popularity of erasure coding in cloud storage architecture and peta-scale computing, we present a new block layout mechanism (FINGER) for Erasure coded HDFS. Along with the new block-layout policy in HDFS, we also propose to use multi-threaded encoding and decoding to improve their respective performance. In addition to this, we change HDFS to allow update operations on data blocks. While the researches in erasure coding in Hadoop focus on using new erasure codes or changing the scheduling policy of the Map-reduce framework, we focus on the changing the underlying block placement policy. Since our design focuses on the block-layout, we can easily apply any erasure codes to HDFS and observe their benefits in addition to the better write and update throughput obtained with FINGER.

From the motivating example in Section 4.1, we see that choice of correct block size reduces the amount of data being written into the erasure coded HDFS, while the files are written into or updated. In this chapter, instead of changing the block sizes we perform chunking of blocks into smaller sub-blocks. The number of chunks that are produced from each block is dependent upon the erasure code being used.

### 4.2.1 System Architecture

In this sub-section, we describe in detail our proposed design. Our primary design goal is to reduce the amount of I/O necessary to perform re-encoding when the contents in any blocks are changed in erasure coded HDFS. Our design goal follows two intuitions:

- *Perform erasure coding inside each block only*. If we perform erasure coding across different blocks, changing the contents of the individual blocks/files requires re-encoding of the blocks, which in-turn causes extra disk/node I/O

- *Reduce the effective block size of Erasure coded HDFS, while keeping the same meta-data size.* If we reduce the block size for erasure coding inside the large block, we in-

crease Namenode's metadata size, which limits the maximum number of files HDFS can support.

Figure 35 illustrates the basic architecture of the proposed FINGER design. In FINGER, the client interacts with the Datanodes via Namenode. Figure 35 details a typical case of an HDFS cluster with 6 Datanodes, a Namenode and a FINGER client. In contrast to HDFS-RAID, FINGER client performs on-the-fly erasure coding. Once the client receives a block, it chunks the blocks into smaller sub-blocks and performs erasure coding across these chunks/sub-blocks. Other incoming blocks are also encoded based on the erasure code in use. The chunks/sub-blocks are appended according to Algorithm 1, which shows the chunking and layout mechanism. In Figure 35 the client has 4 blocks: A, B, C and D. All of this data is forwarded to the FINGER client. As soon as the FINGER client receives these blocks, they are chunked into chunks/sub-blocks based on the erasure code in use. In this particular case, RS(6,4) code is used and A is chunked into A1, A2, A3 and A4, B into B1, B2, B3 and B4, and so on for C and D. After chunking, the chunks from the same block are erasure coded to create parity chunks ((P1,Q1 from A1, A2, A3, A4), (P2,Q2 from B1,B2,B3,B4), and so on). Then, after 6 block locations are determined from Namenode, chunks/sub-blocks A1, A2, A3, A4, P1 and Q1 are written to these locations. As soon as B1, B2, ..., Q2 become available, they are appended with A1, A2, ..., Q1 respectively. The chunks from C are appended to chunks from B and chunks from D are appended with chunks from C.

### 4.2.2 Block Chunking and Block-Layout

The key challenge here is how to determine the right chunk size to perform erasure coding and how to place the chunks in blocks, so that we do not increase the metadata size. With the fixed layout (as described in sub-section 4.2.1), we can easily determine the chunk locations of B/C/D blocks from the chunk locations of A, the original block size,

and erasure code being used. Based on the erasure code used and the type of request, i.e., write/update, the FINGER client performs erasure coding and decides if the data is written to the new block, appended with another block, or in-place updated.

**Algorithm 1** Block-Chunking and Layout Algorithm

1: $k \leftarrow$ length of *data stripe*
2: $m \leftarrow$ length of *parity stripe*
3: Chunk each data block into $k$ separate chunks
4: Encode each block using Algorithm 2
5: Write first block's chunk contents to $k$ *new-block*s
6: Write first parity block's chunk contents to $m$ *new-parity-block*s
7: **for** $i$ = 2 to $k$ **do**
8: Append $k$ chunks of the $i^{th}$ block to *new-block*s
9: Append $m$ parity chunks of the $i^{th}$ block to *new-parity-block*s

Let's assume that the erasure code being used is $(n, k)$ erasure code, where $k$ is the number of data blocks and $m = n - k$ is the number of parity blocks to be generated after encoding. For each block, the FINGER client chunks the large block into $k$ smaller chunks/sub-blocks. If the request is a write operation, the FINGER client performs erasure coding on the first incoming block and produces $m$ parity chunks. It then sends $k + m$ chunks to be written to Datanodes as new blocks. When another block is written by the client to the FINGER client, it performs erasure coding on this new block. Instead of writing the new-chunks to new blocks, FINGER appends these chunks to the previous incomplete blocks from previous block chunking. As parities generated by each block's erasure coding are equal to sub-block sizes, the parities are also appended until the parity-block becomes full. This process continues until $k$ blocks are erasure coded. Now the $(k + 1)^{th}$ block being written to HDFS is erasure coded and written into a new block and

the $(k+2)^{th}$ block will be appended.

HDFS and HDFS-RAID both necessarily overwrite an entire file when any change, small or large, occurs. In contrast, we relax this condition by allowing the update size to be a multiple of the individual block size. This eliminates the need to re-write the large file for small changes in the file, thus we can improve the system's small write/update performance. In a FINGER client the update granularity is block-size. In addition to allowing update operation, the blocks are not immutable in FINGER, as they are in HDFS and HDFS-RAID. FINGER can seek to a certain block location and change the contents of that block. The smallest seek granularity for writes are chunk/sub-block size, which is determined by the erasure coding parameters and the *block size* set by the client.

When a client provides new content for a file, FINGER looks into its Namenode's metadata information, determines the blocks used for erasure coding and the block number being updated. FINGER then erasure codes the new block and generates the parity chunks. Based on the block number, default block size, and block locations FINGER determines the seek location and seeks to that location for updating the content. Let's say the block size is 128 MB, the erasure code being used is RS(6,4) code, the file has 8 blocks and we are updating the $3^{rd}$ block. From the erasure coding information in Namenode's metadata, we know that the $3^{rd}$ block is combined with block 1, 2 and 4 to perform erasure coding. Note that the original HDFS-RAID also needs to keep this information in Namenode; as such, FINGER does not increase the metadata size w.r.t. HDFS-RAID. Now, the four erasure coded blocks have a single entry in their block locations and the chunk locations can be inferred from this information. To update block 3, we first seek to the beginning of all the 4 blocks (In Figure 35, it is the beginning of A1, A2, A3 and A4) then we compute the offset using *Seek_offset* = *block_number* × *chunk_size*. We then seek using this offset and then perform new chunk/sub-block writes at this location. Since the chunk's sizes are fixed for one instance of erasure-coded Hadoop, the new content does not overwrite the adjacent

block's contents.

FINGER can also perform read requests in parallel, because a single block is stripped across multiple Datanodes. The disadvantage of this method is that the client needs to open streams to each of the nodes at specified offsets and then performs reads. One might argue that for multiple block read requests, there are fragmented reads; we would like to point out that the fragmentation is not huge. For random block access, a maximum of additional seek is equal to $(k-1) \times chunk\_size$ w.r.t. HDFS-RAID or HDFS. The FINGER client appends these sub-blocks together and forwards to the user-client. From user client's perspective, the block size has not changed at all.

### 4.2.3 Erasure Coding

In HDFS-RAID [3], erasure coding is strictly a serial procedure because of the erasure coding granularity. The erasure coding is performed in the granularity of blocks, i.e., HDFS-RAID does not perform erasure coding within a single block, but needs to wait to receive $k$ data blocks from Datanodes to Raid-Node and then performs erasure coding. Since the data is first replicated and then erasure coded in the background by Raid-Node, the erasure coding consumes a lot of network and disk bandwidth.

**Algorithm 2** Multi-threaded Encoding Algorithm

1: $k \leftarrow$ length of *data stripe*
2: $m \leftarrow$ length of *parity stripe*
3: $t \leftarrow$ Total no. of suported hardware *threads*
4: **if** $m < t$ **then**
5: New threads for encoding $\leftarrow m$
6: **else**New threads for encoding $\leftarrow t$
7: Encode all $k$ blocks in parallel using encoding threads

In contrast to HDFS-RAID, FINGER performs erasure coding at the granularity of chunks/sub-blocks. Performing erasure coding within each block allows parallel encoding of the blocks. Although each block can be erasure coded independently, we limit the number of encoding threads according to Algorithm 2. If we spawn more software threads than the number of hardware threads, the CPU needs to perform scheduling for each thread, which reduces the performance of individual threads. Thread creation also introduces overhead in memory and the CPU, and if a very large number of threads are created, thread creation time dominates the execution time effectively worsening the performance. So, the number of threads spawned is set to be never greater than total number of hardware threads in the system. If the data stripe length is less than the number of hardware threads, we only spawn threads equal to the data stripe length. Once the threads are created, we encode each block in parallel and write the contents to Datanodes because all the blocks are independent of each other. Since erasure coding is done at the chunk/sub-block granularity, even if a file has a single block it is a candidate for erasure coding and we save both network and disk bandwidth.

### 4.3 Evaluation

To validate the practicality of FINGER and it's block-layout mechanism with multi-threaded encoding and decoding, we implemented it in HDFS release 0.22.0 and deployed the HDFS cluster running 10 Datanodes and 1 Namenode. One of the machines running a Datanode also runs a FINGER client and a user client. All the nodes are Linux-based machines with two 2.30GHz Intel(R) Xeon(R) CPU E52630 processors, 64GB RAM and 320GB hard drives. Each DataNode is equipped with an Ethernet interface card with a network speed of 1Gbps. The physical entities are connected over a 24-port HP 1810-24G switch. We use "*Reed-Solomon Code*" for erasure calculations. While the experiments are performed with RS(6,4) and RS(10,8) erasure code, FINGER design can also be easily

extended to use any erasure code.

In this section, HDFS refers to the system using 3-way replication. HDFS-RAID, by default, is used as a middle-layer to perform erasure coding. Originally, data is 3-way replicated; the HDFS-RAID then reads a copy of the data, performs erasure coding, writes the parities, and reduces the replication level to 1. For the sake of fair comparison, we change the HDFS-RAID to perform online erasure coding, i.e., data is erasure coded as soon as $k$ blocks of data are available. The data blocks are forwarded to data nodes as soon as they are available, but a copy is kept in memory to calculate parity data. If $k$ blocks of data are not accumulated within 1 minute, the data blocks are triplicated. In the following experiments, we set the data arrival rate high enough to perform online erasure coding, i.e., using HDFS-RAID none of the blocks are 3-way replicated. During our experiment, we choose a file of $10$GB size as a reference file. One might argue that it resides in memory itself and does not get flushed to disk, thus biasing the results. Since each of the nodes has 64GB of RAM, even if the file is triplicated and kept in a single data node, it still fits in memory. Thus, the comparison between HDFS, HDFS-RAID and FINGER is fair. [84] did an extensive evaluation on the trade-off of erasure coding in the MapReduce framework and showed that erasure codes can reduce the execution time of analytic jobs like *Sort*, *WordCount* and *CloudBurst* by up to $51\%$. Thus, we do not evaluate the run-time of MapReduce jobs in this chapter.

### 4.3.1 Write-Performance

We first evaluate the impact of block sizes on the write performance. Figure 36 shows the write throughput vs the block-size for HDFS, HDFS-RAID and FINGER. We observe that with the increase in block size, there is an increase in the write throughput for all three. This is due to the increase in sequential writes, which is caused by the increase in the block size (sub-block size for FINGER, e.g. 32MB block = 8MB sub-block,

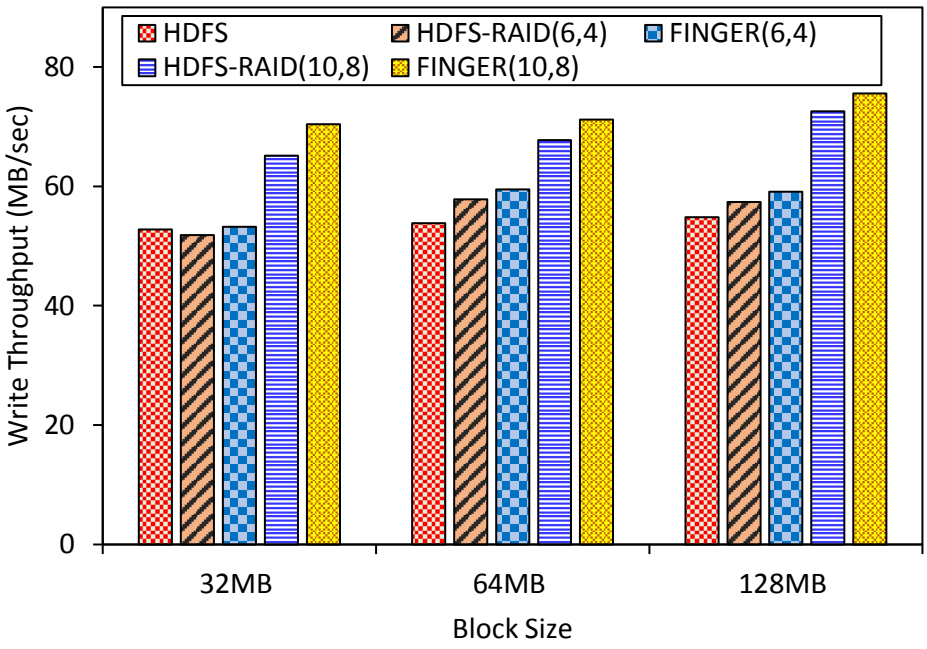



Fig. 36. Write throughput for a 10GB file-write with various block-sizes.

while 128MB block = 32MB sub-block). In Figure 36 and all up-coming figures, HDFS-RAID(6,4)/FINGER(6,4) represents the use of RS(6,4) code in HDFS-RAID/FINGER, i.e., 4 data blocks are used to generate 2 parity blocks. Similarly, FINGER(10,8) computes 2 parity blocks from 8 data blocks.

From Figure 37, we can see that RS(6,4) erasure coding reduces the amount of disk I/O by 50% for writes. In 3-way replication, for every write request of 4 blocks, we need a total of 12 block writes, while HDFS-RAID and FINGER require only 6 block writes, i.e., we halve the disk-write traffic. This significant reduction in write-traffic is one of the main reasons behind the increase in write-throughput for HDFS-RAID and FINGER. RS(10,8) also reduces the write traffic by $58.33\%$ and we observe better write throughput.

In addition to the reduction in the write traffic, FINGER performs erasure coding in parallel, which further reduces the encoding time and effectively improves the write throughput. Taking into account block-striping overhead, thread-creation overhead and era-

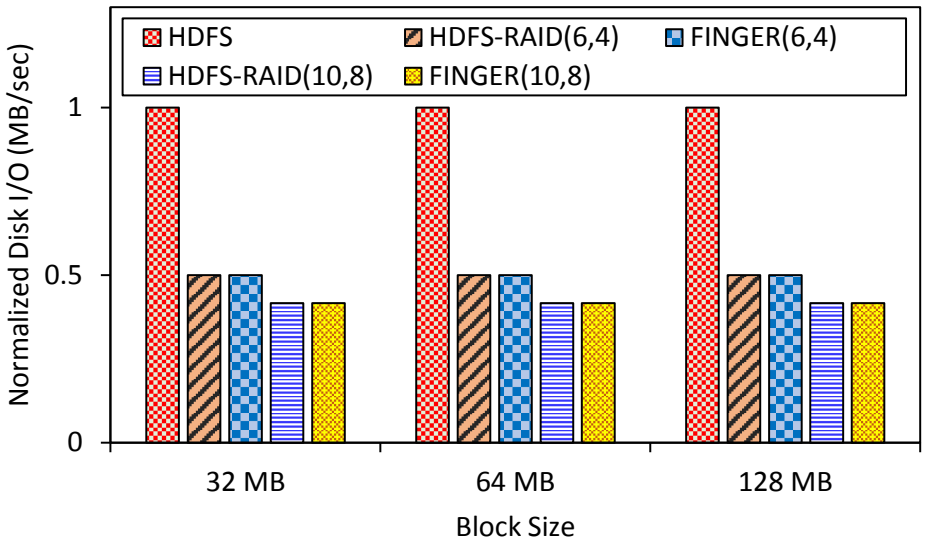



Fig. 37. Disk I/O for a 10GB file-write, normalized to 3-way replication.

sure coding, FINGER can improve write performance by up to $38.20\%$ w.r.t. 3-way replication and by up to $8.08\%$ w.r.t. HDFS-RAID. FINGER(10,8) has better write throughput than FINGER(6,4) because FINGER(10,8) encodes using 8 threads while FINGER(6,4) utilizes only 4 threads.

### 4.3.2 Update-Performance

We also evaluate the performance of HDFS, HDFS-RAID and FINGER during a file's content modification. We limit the content modification size to be a factor of block sizes. In our experiments, updating 128MB of data translates to 4 consecutive blocks for 32MB block size, 2 blocks for 64MB block size, and 1 block for 128MB block size.

In 3-way replication, updating a block's content is the same as replacing the old block with a new block and updating the replicas. In Figure 39, we normalize the disk I/O with 3-way replication. When the block size is 32MB, updating 128MB means re-writing the whole stripe for HDFS-RAID(6,4) or half of the stripe for HDFS(10,8). Since the contents of the blocks are modified, we need to read the data from remaining blocks in the same

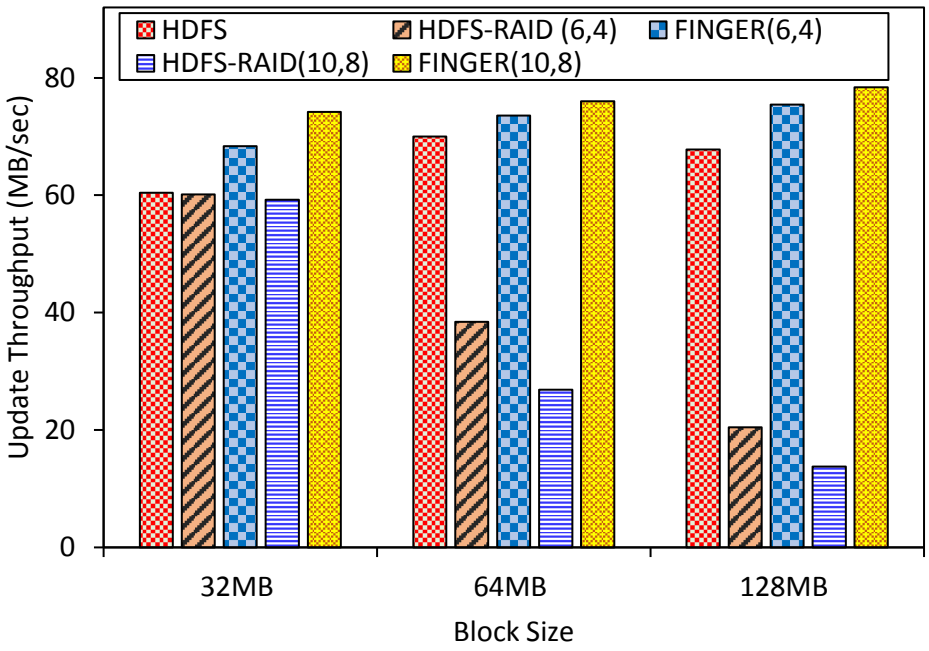



Fig. 38. Update throughput for updating 128MB data in a 10GB file with various block–sizes.

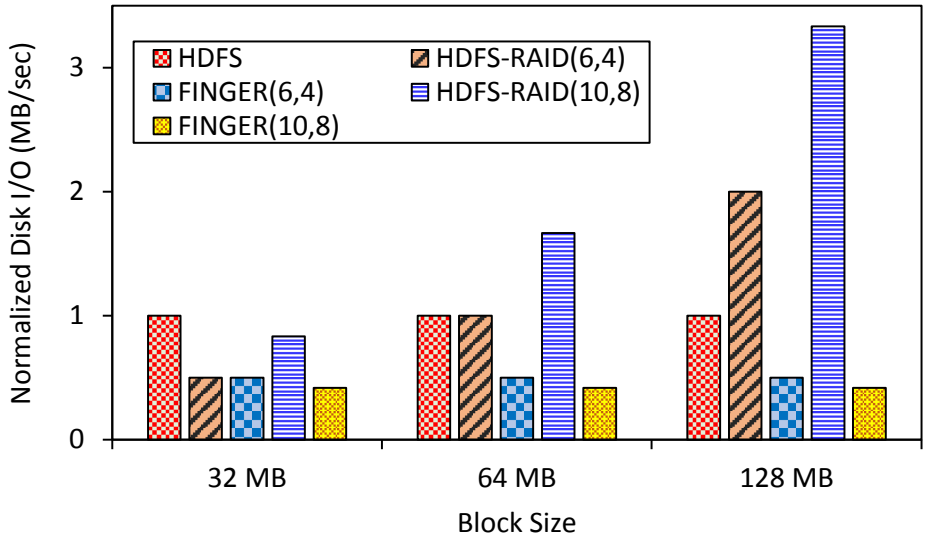



Fig. 39. Disk I/O for a 128MB data update in a 10GB file, normalized to 3-way replication.

stripe for parity re-computation. Therefore, HDFS-RAID(10,8) with 32MB blocks requires $4 \times 32 = 128$MB extra reads. While the disk I/O is less for HDFS-RAID, the computational

overhead caused by encoding keeps the update throughput very similar to the default HDFS (see Figure 38). With FINGER, the encoding is performed at sub-block granularity and does not incur any extra I/O regardless of the block size. HDFS-RAID needs to read other blocks in the erasure coding stripe.

The problem with HDFS-RAID becomes aggravated with an increase in block size and stripe length. For 128MB block size, HDFS-RAID(10,8) has a write throughput of around 13MB/sec. This very low update throughput is the result of performing encoding at the block level granularity. When updating a single block (128MB), it requires the extra reads of 7 blocks ($7 \times 128 = 896$)MB. This huge volume of extra disk I/O significantly reduces the update throughput. From Figure 38, we conclude that FINGER outperforms HDFS-RAID(10,8) by up to $5.68\times$. Although disk I/O for an update is reduced by $58.33\%$ w.r.t. HDFS, the observable update performance improvement is only $8.6\%$ because of the overhead of encoding and multi-threading.

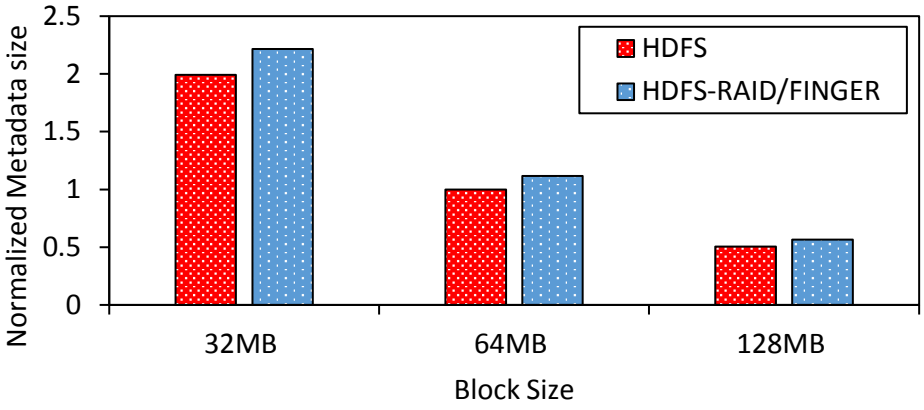



Fig. 40. Normalized Namenode's Metadata Size for a 10GB file with various block-sizes.

One might argue that keeping a small block size eliminates the need for FINGER because small blocks do not incur extra disk I/O during update operations. However, we would like to point out that all the metadata information of the deployed system is kept in Namenode's memory. Figure 40 shows the metadata size for 10GB file. The Figure is

normalized to HDFS's Namenode's metadata size (block size is 64MB). We observed that reducing the block size by half almost doubles the metadata size. The increase in metadata size limits the number of files the HDFS cluster can support and becomes a scalability bottleneck. This is the reason behind large block sizes in Hadoop. Since FINGER does not introduce any metadata entries, the metadata size is identical to that of HDFS-RAID, which is slightly larger than HDFS due to the extra information stored by erasure coding and parity blocks.

### 4.3.3 Read and Recovery Performance

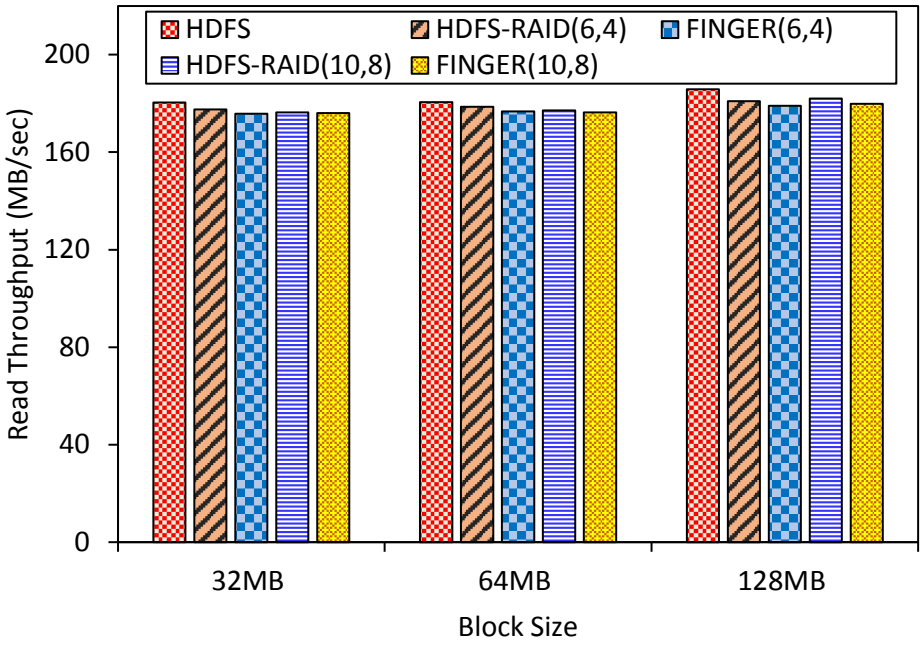



Fig. 41. Read throughput for a 10GB file-read with various block-sizes.

We also evaluate the read throughput and recovery throughput of the three systems. Figure 41 shows the read throughput for the 10GB file that was written in the previous experiment. This Figure shows the read-throughput when no node failures were present. If systematic codes such as RS code are used for erasure coding, only parity data is generated;

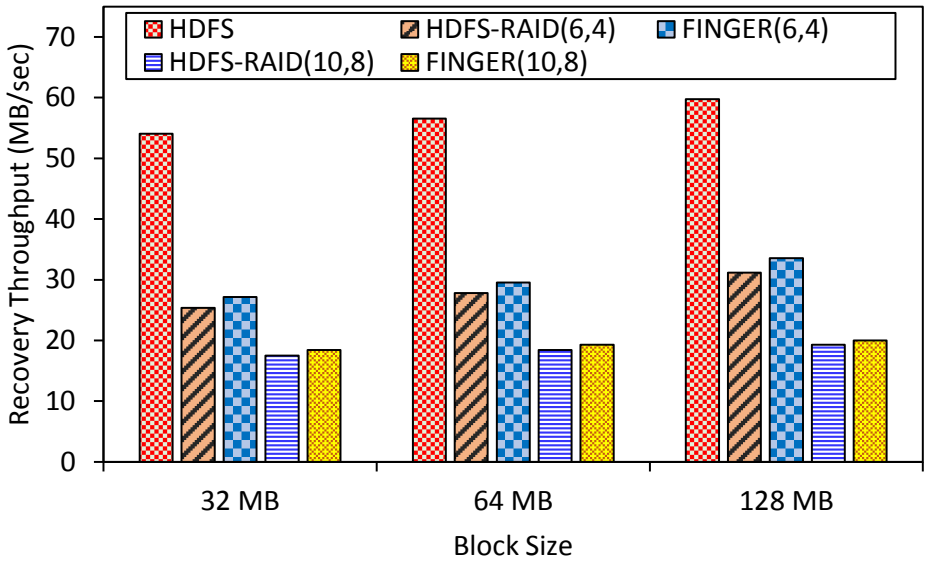



Fig. 42. Recovery throughput for data reconstruction after a single node-failure.

without changing the contents of the original data. If there are no failures in the system, the performance is very similar between HDFS, HDFS-RAID and FINGER due to one exact-copy of data being available.

The penalty of using erasure coding is decreased performance during recovery w.r.t. replication. When some of the nodes fail, the data needs to be reconstructed or recovered. In replication, the clients can simply switch to using another node containing the data. In an erasure coded system, the data needs to be recomputed by reading data from operational nodes. Extra reads from the surviving nodes and re-computation degrades the performance of HDFS-RAID and FINGER (see Figure 42). Though FINGER performs worse than replication during recovery, it performs parallel decoding and has better recovery throughput than HDFS-RAID (around $6.9\%$ improvement).

## 4.4 Summary

In this chapter, we proposed a new block-layout mechanism for erasure-coded HDFS. HDFS performs better with large block size because of the sequential data access along

with the relaxed metadata management in the Namenode's memory. While default HDFS favors larger block size, erasure coding across blocks favors smaller block sizes because updating multiple blocks in a single stripe requires less extra I/O bandwidth than updating a single block. To address this opposing requirements of HDFS and erasure-coding, we proposed and implemented FINGER, which chunks a large block into smaller chunks/sub-blocks, performs erasure coding across chunks of the same block, and determines the layout of these chunks into larger blocks. This deterministic layout allows us to keep the metadata size the same as the original system, i.e., HDFS-RAID, but observe significant improvement in update performance. FINGER performs multi-threaded online erasure coding, in contrast to HDFS-RAID, which does replication in the foreground and sequential erasure coding in the background. To demonstrate the effectiveness of FINGER, we implemented it in HDFS and evaluated the read, write, and update throughput for 10GB file reads, writes, and 128MB update operations. Our scheme was found to improve the write and update performance by up to $32\%$ and $8.6\%$ w.r.t. 3-way replication and up to $5.02\%$ and $5.68\times$ w.r.t HDFS-RAID respectively, while maintaining a similar read throughput as 3-way replication for the case of no node failures. During node-failures, our scheme was able to improve the recovery rate by $6.9\%$ w.r.t. HDFS-RAID.

# CHAPTER 5

# COARC: CO-OPERATIVE, AGGRESSIVE RECOVERY AND CACHING FOR FAILURES IN ERASURE CODED HADOOP

In this chapter, we recognize that lack of data sharing across various application during the degraded read process reduces both the performance and interoperability. We propose CoARC mechanism to perform pro-active data recovery during degraded reads efficiently and cache these data blocks . In addition to eliminating the degraded-read of the same data multiple times, it also reduces the node repair time if the transient failure is later identified as the permanent failure.

## 5.1 A Motivating Example

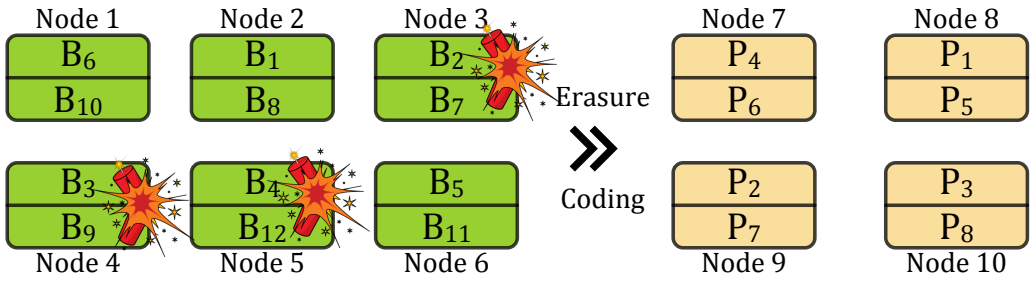



Fig. 43. A 10-node Hadoop cluster with 12 native blocks and 8 parity blocks. We assume the use of (10,6) erasure code and 3 nodes fail while Hadoop client is running. Blocks 1 to 6 form one data strip and Blocks 7 to 10 construct another data strip.

In this section, we explain via a motivating example why the default data recovery process of HDFS during temporary failures causes huge overhead to network resources and the overall program execution time. This section illustrates the shortcomings of the default degraded data read mechanism in presence of multi-node failures and multi-client

scenarios.

Figure 43 illustrates an erasure coded Hadoop cluster with 10 data nodes, but hides the NameNode and other components (like network switches) connecting these nodes. We assume that a file comprises 12 blocks, namely $B_1$ to $B_{12}$ and is distributed uniformly across the cluster. HDFS-RAID reads these blocks from Node 1 to Node 6 and computes parities and places them across Nodes 7 to 10. Here, we assume that (10,6) erasure code is used, which is a 4-node fault-tolerant erasure code. For load balancing purposes, the data and parity blocks are distributed evenly across all data nodes. Thus, each of these nodes has 2 data blocks. Blocks $B_1$ to $B_6$ form one data strip and $B_7$ to $B_{12}$ form another strip. Parities $P_1$ to $P_4$ are part of first strip and $P_5$ to $P_8$ are parity blocks of second strip.

Consider a case, when nodes 3, 4 and 5 fail. During this failure period, when a client wants to read the file in HDFS, the client needs to perform degraded read for $B_2$, $B_3$, $B_4$, $B_7$, $B_9$, and $B_{12}$. To recover from any number of failures, we need to read $k$ blocks of data in an $(n, k)$ MDS erasure code. Thus, the client reads all other data blocks in the same strip during the degraded read process, decodes the failed data block, and serves the request. Since this request is a whole file read, the data blocks are read from $B_1$ to $B_{12}$ sequentially in Hadoop. $B_1$ is read first and a block missing exception is encountered for $B_2$-read. Hadoop subsequently goes through a time-out mechanism and performs multiple retries for block access to realize that $B_2$ is temporarily unavailable. HDFS then initiates the degraded read process.

During the degraded read, $B_3$ and $B_4$ are also identified as unavailable blocks. The client then recovers $B_2$ by reading $B_1$, $B_5$, $B_6$, $P_1$, $P_2$, and $P_3$. Once the read request is fulfilled, the recently recovered data block, i.e., $B_2$ is discarded by the client because this failure is classified as temporary. The main problem of Hadoop becomes apparent during the access of $B_3$. When $B_3$ is requested, the client needs to go through the timeout mechanism again to realize that $B_3$ is a failed block. In Hadoop, the recently recovered $B_2$

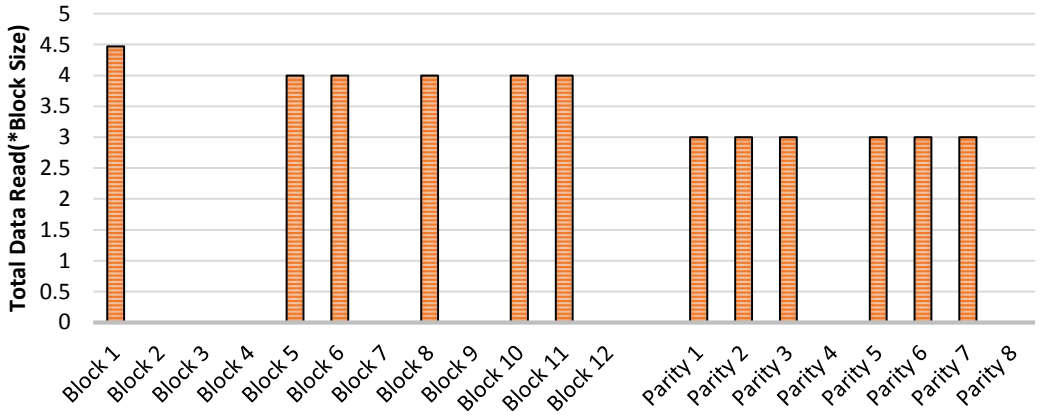



Fig. 44. Amount of data read from each data block during triple node-failures. The erasure code used is RS (10,6) and block size is 64MB.

is discarded, and it doesn't record failure information for unavailable blocks encountered during past degraded reads. Thus, the client needs to re-perform failure detection for $B_2$ and $B_4$ to identify the block unavailability in the data strip. This process is repeated for both $B_4$ and the second strip, which causes the client to spend most of its execution time in failure identification phase via timeout mechanism.

As the degraded read mechanism recovers only the requested data block, the client needs to read the same data blocks multiple times for recovery from multi-block failures. We ran a simple file-read test in HDFS-RAID, to illustrate the above example. We injected three node failures, and we recorded the amount of data read from each block. With the presence of node failures, Figure 44 illustrates *read amplification*, which is the increase in total data transferred (caused by the data recovery process), observed in our test. We can see that $B_5$ (Block 5), $B_6$, $B_8$, $B_{10}$, and $B_{11}$ are accessed 4 times. Out of this 4 block accesses, 3 reads are for recovery of three failed blocks in a strip. This can be verified by looking at the access of parity blocks, i.e., parities 1, 2, 3, 5, 6, and 7 are read thrice. $B1$ has more accesses than other blocks because it is the starting block for strip, and a portion of $B_1$ is read during the degraded read. When other block failures are detected during

this degraded read of $B_2$, the recovery stream is rebuilt to mark the failure location. This stream rebuilding process increases the accesses for the first block in each strip. For the second strip, since $B_7$ is the failed block, no such increase is observed. In our test, we observed that the system spent $357$ seconds to read the whole file containing $12$ blocks, and each block size was $64$ MB. Out of this time, the time spent for failure identification was $334$ seconds (which is around $93\%$ of total execution time), although a block failure identification requires around $18$ seconds. This illustrates the shortcoming of lacking failure remembrance during degraded reads, which causes system to spend most of the time for identifying the same failure repeatedly, and for redundant reads of the data blocks. In case of multi-client scenario, multiple clients access a failed data block, and the same data block is recovered independently by different clients, and all the clients perform failure identification, which increases the network resource consumption and program execution dramatically.

## 5.2 CoARC Design and Analysis

In this section, we present the design of the Co-operative, Aggressive Recovery and Caching (CoARC) for Hadoop, whose main idea is to recover from all block failures identified during the degraded read issued by the Distributed File System Client, and cache the regained data. Its primary objective is to eliminate the redundant failure identification and excessive recovery of data blocks in the cases of multi-node-failure or multi-client data access. It mainly focuses on three key designs: (i) Recover all the unavailable data blocks in the same data strip during degraded reads, (ii) Instead of discarding the recovered data after serving degraded reads, cache them so that other clients do not need to perform re-recovery, and (iii) Efficiently evict the data blocks, based on the node failure information and strip distribution, from the shared cache. This section also presents the reliability analysis and mathematical analysis on potential gains in reliability and performance by using CoARC.

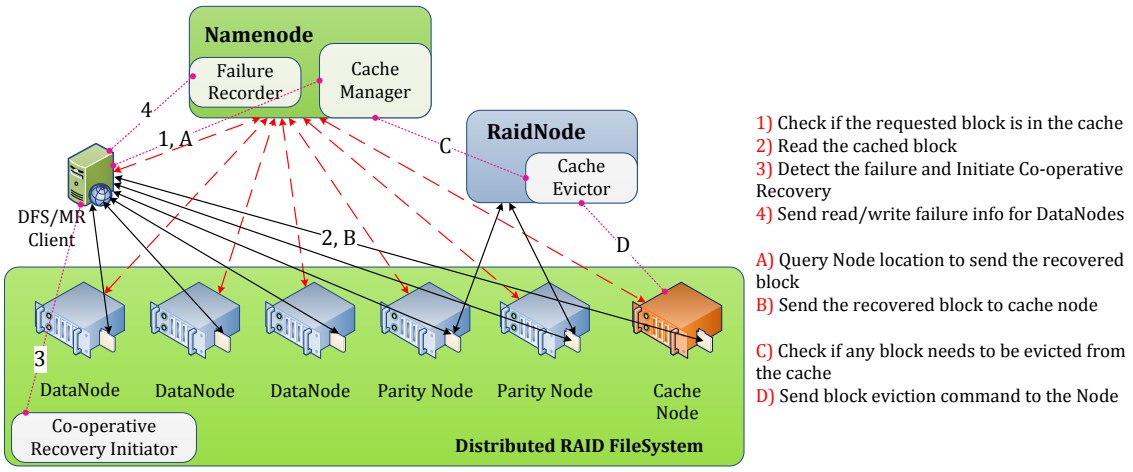



Fig. 45. System architecture of CoARC detailing the data read and recovery process. The dotted lines with arrows represent Heartbeat messages to NameNode; the solid lines show the data/block transfer, and dotted line without arrow-ends are the new communication channels introduced to HDFS.

### 5.2.1 Design Goals

The primary goal of CoARC is to reduce the degraded read traffic and degraded read time for multi-node failures and multi-client access. The design aims to achieve:

- *Eliminate the redundant block failure identification*. The lack of failure sharing and discarding of recovered data will cause the same client or other clients to re-identify the block as unavailable to trigger the degraded read process. This redundant failure identification increases the program execution time extensively.

- *Eliminate redundant recovery of failed blocks*. If only the requested block is recovered by a client, and this recovered data is not shared with other clients, the unavailable block will be recovered by all clients independently and causes redundant read access to other data blocks in the recovery strip.

### 5.2.2 Design Overview

As discussed previously, the idea of CoARC is simple. To tackle the redundant recovery of failed blocks, CoARC stores the recently recovered data blocks in the cache space and serves all incoming requests via this cache. To further reduce the network consumption, CoARC not only recovers the requested data block, as it is done in traditional degraded read mechanisms [86], [50, 46] but also recovers all unavailable blocks in the failed strip. The architecture of CoARC is illustrated in Figure 45. As depicted in the figure, we introduce various modules in different layers of Hadoop. Figure 45 also details the data read and recovery process. We briefly describe each of these modules below:

**Co-operative Recovery Initiator:** This module is responsible for identifying the failed blocks in the strip before initiating the degraded read process. In default HDFS-RAID [3], when a block read request is initiated and the block is unavailable, degraded read is performed only for the requested block. The co-operative recovery takes the degraded read process one step further. When any block is identified as unavailable, before the degraded read request is initiated, this module checks the availability of all blocks spanning the strip by performing $4KB$ reads from the block start offset. If any other blocks are identified as unavailable blocks, they are marked as to be recovered, even if they are not requested by the client. Thus, we can say this module initiates an aggressive recovery mechanism, i.e., all blocks classified as unavailable blocks during our enhanced degraded read process are recovered immediately, even if the original read request doesn't include the other block. Irrespective of the number of block failures in a strip, we always need to read a whole data strip worth of data for recovery, i.e., $k$ blocks in an $(n, k)$ MDS erasure code like RS code [60]. CoARC takes advantage of this property and performs Co-operative recovery of all the unavailable blocks in the strip without incurring any overhead in the network traffic. We perform this aggressive recovery because MapReduce applications

typically read the whole file during the Map phase.

**Cache Manager:** This module is augmented to the NameNode in Hadoop to keep track of blocks available in the cache. When any client wants to perform data read, it checks with the NameNode for the block location. If the requested block is in the cache, it advises the cache locations rather than the primary locations to the client. The client can then perform read from the new cache location, instead of the original location. There is no cache consistency issue in CoARC because all blocks are marked as immutable in Hadoop. In addition to this, the module also interacts with the Cache Evictor module running in the RaidNode, to evict blocks if there are more blocks in the *Cache Node* than specified by the system administrator (for details see subsection 5.2.3).

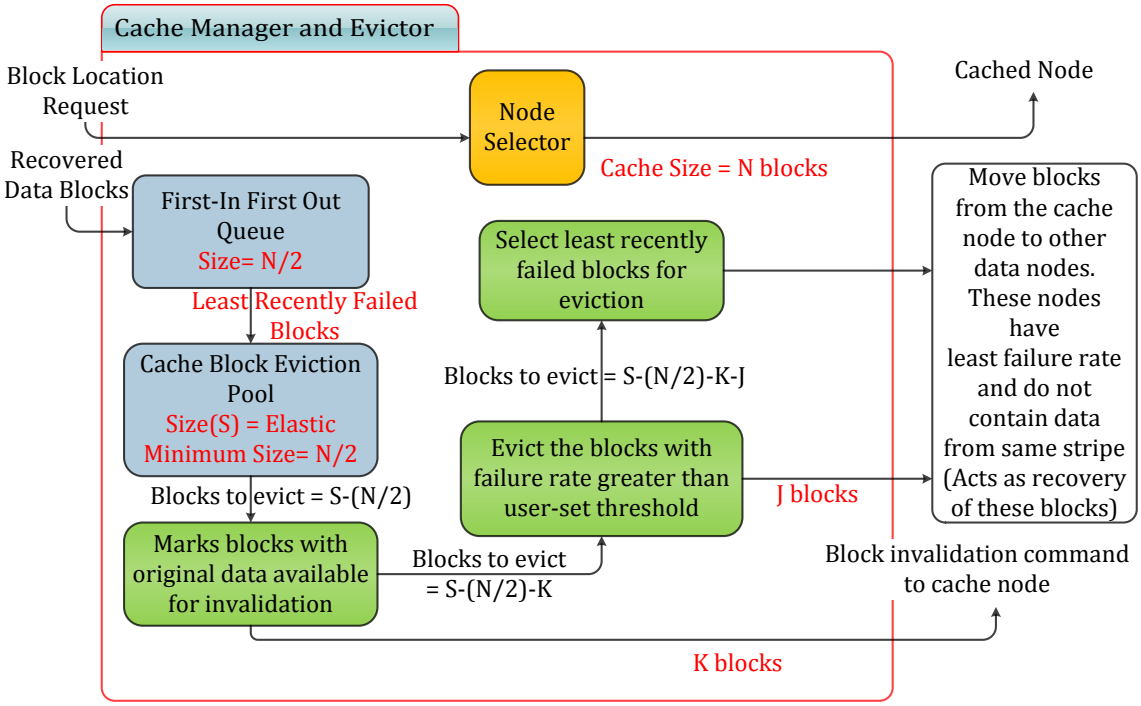



Fig. 46. Coalesced view of the Cache Manager and Evictor. It also details how the decision for block eviction is made. The cache in CoARC is an elastic cache, and Evictor periodically performs block eviction.

**Cache Evictor:** Figure 46 details the inter-operation inside the Cache Manager and

Cache Evictor. In CoARC, the system administrator has the capability of setting the cache size, i.e., number of blocks that cache will hold. The cache in CoARC is an elastic cache, i.e., at any point the cache can have more data blocks than the specified size, and the maximum blocks it can hold is determined by the storage space in the Cache Node. The cache evictor performs periodic eviction of blocks from the cache. Thus, the cache used in CoARC is different from traditional caches. The decision on which block to evict is detailed in Figure 46. The data blocks evicted from the cache are either invalidated or written to other data nodes, and this decision is based on the type of the failure, i.e., permanent or temporary block failure.

**Failure Recorder:** The failure information for every node is recorded in the failure recorder. The number of entries in the record, which tracks the node failure rate, is equal to the number of data nodes in the cluster. The failure-rate is basically a counter to record how many times read requests to blocks in that node was unavailing. This module also maintains another record for the number of times a request for specific block was unsuccessful. Any data block, which encounters an I/O exception, is entered on this record for block failure rates. If another request for the same block fails, the counter is increased in block-failure record. Since this record is cleared periodically, and the duration can be set by the administrator based on how much history he/she wants to maintain, there is no exponential growth of the record. Although the failure information for nodes and data blocks is utilized only for recovery cache in this work, the failure information can also be used to make better decisions on scheduling the MapReduce tasks to reduce the task failure rate.

### 5.2.3 Least Recently Failed Cache Replacement Algorithm (LRF) for Hadoop

CoARC introduces a recovery cache for Hadoop. This cache is different from traditional caches, which performs eviction based on block access frequency like LRU [23] or FBR [61]. Only blocks that are recovered by the client during degraded reads are stored

in the cache. The block eviction is based on the Least Recently Failed Caching Algo-

**Algorithm 3** Least Recently Failed Caching Algorithm

**Input**: Recovered blocks $b_1, b_2, b_3, ..., b_i, ...$
**LRF_Cache**$(b_i)\{$
1: Write the block to cache node;
2: Put $b_i$ to the head of $LRP$;
3: Move $b_k$ to the head of $EBP$ and update $S$, by $S = S + 1$ ▷ $b_k$ is the tail of $LRP$
$\}$
**LRF_Evict**$(N_e)\{$
1: **if** $N_e > 0$ **then** ▷ Available blocks are to be evicted
2: **for** Each block $(b)$ in EBP **do**
3: **if** it is alive in the underlying storage **and** $N_e$ 0 **then**
4: Invalidate $b$ in cache
5: Remove $b$ from $EBP$
6: $N_e = N_e - 1$, and $S = S - 1$
7: **if** $N_e$ 0 **then** ▷ Most-frequently failed blocks are to be evicted
8: **for** Each block $(b)$ in EBP **do**
9: **if** If failure-rate is greater than the threshold **and** $N_e$ 0 **then**
10: Move $b$ from cache to underlying storage
11: Invalidate the old $b$ in underlying storage
12: Remove $b$ from $EBP$
13: $N_e = N_e - 1$, and $S = S - 1$
14: **while** $N_e$ 0 **do**
15: Move $b$ from tail for $EBP$ to underlying storage
16: Invalidate the old $b$ in underlying storage
17: Remove $b$ from $EBP$
18: $N_e = N_e - 1$, and $S = S - 1$
$\}$

rithm, which is shown in Algorithm 3. The idea behind evicting the least recently failed block from the cache is to allow more time for the recently failed nodes or unavailable blocks in the underlying storage to come back alive because most of the node failures are temporary/transient [24].

When blocks are evicted from the cache, we store the data to those nodes, which do not have data from original parity strip and have least failure frequency. The failure frequency is obtained from the failure recorder in NameNode. The advantage of LRF is that we don't incur huge write back to the HDFS nodes because data nodes often come back alive after

some time. Even if nodes don't come back alive, since we already recovered the data, the amount of data to be recovered by the recovery thread (not the client) of HDFS is reduced. Thus, we also eliminate redundant recovery of the data blocks, when permanent node failures occur.

We conduct reliability analysis and a simple mathematical analysis to compare the runtime of default HDFS-RAID and CoARC in terms of the time taken to service a block read. Our goal is to provide preliminary insights into the potential benefits of CoARC. For our analysis, we assume a cluster with $N_n$ homogeneous nodes. The cluster is protected by $(n, k)$ erasure code, i.e., fault tolerance $t$ is $n - k$ concurrent node failures.

### 5.2.4 Analysis

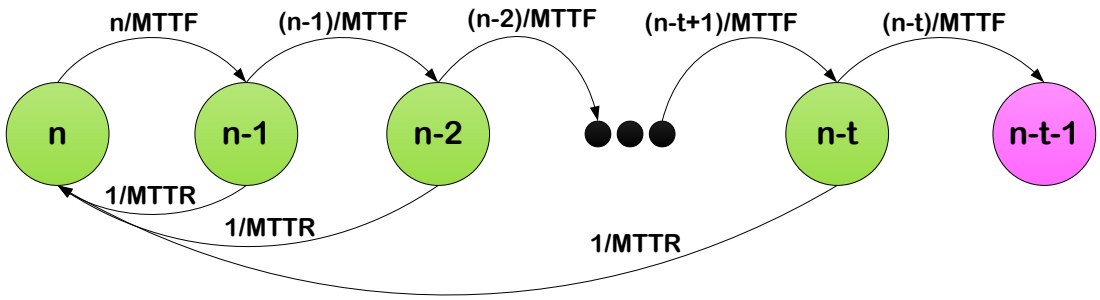



Fig. 47. Markov Model for Failures in Erasure Coded Storage. The storage consists of $n$ nodes and employs $t$ fault tolerant erasure code. MTFF is a constant and MTRR can be computed by using Equations 5.1 and 5.2.

**Reliability Analysis**: To analyze Mean Time to Data Loss (MTTDL), we introduce a standard Markov chain model [16, 39] for a data strip in erasure coded cluster with $t$ fault tolerance in Figure 47. As mentioned in previous sections, the total repair time consists of failure detection and block repair time. If we neglect the processing time, the block recovery time is given by $\frac{(n-t)*B}{\gamma}$, where $\gamma$ is the repair bandwidth, and $B$ is the block size. In HDFS-RAID, blocks are recovered only when node failures are classified as permanent

Table XIV. Variants in LRF and Explanation

| Variants | Explanation |
|---|---|
| $b$ | A block recovered during degraded read |
| $N$ | Cache size set by the Administrator |
| $LRP$ | Least Recently Failed Pool |
| $EBP$ | Eviction Block Pool |
| $S$ | Elastic size of EBP ($S \geq \frac{N}{2}$) |
| $N_e$ | No. of blocks to evict ($S - \frac{N}{2}$) |

block failures and the degraded read recoveries do not contribute to the cluster or strip reliability. If the system timeout time is $T$ and the block check interval is $t_h$, the MTTR of HDFS-RAID is given by Equation 5.1.

$$MTTR_{hdfsraid} = (T + 0.5 * t_h) + \frac{(n-t) * B}{\gamma} \tag{5.1}$$

$$MTTR_{coarc} = Min((T + 0.5 * t_h), r) + \frac{(n-t) * B}{\gamma} \tag{5.2}$$

In CoARC, the failure detection time is reduced because the data recovered in the degraded read process contributes to the system/strip reliability. If $r$ is the degraded read request rate, the MTTR of CoARC can be evaluated through Equation 5.2. If we consider typical HDFS cluster protected by $RS(10, 6)$ erasure code, where $B$ = $64MB$, $T$ = 30 minutes, $\gamma$ = $1Gbps$ , and $t_h$ = 300 seconds [15], [65], we see that most of the repair time is spent in failure detection. In CoARC, the repair time is also dependent upon the degraded read request rate, and we have a huge potential to improve MTTR. Since the MTTF for individual nodes is usually considered around 4 years [62], the MTTDL for HDFS-RAID

under typical scenarios (considering $t$ = 3) was evaluated to $7.84 \times 10^9$ days. However, if we assume that the degraded read is performed every 30 seconds, CoARC improves the MTTDL to $1.55 \times 10^{15}$ days.

**Performance Analysis**: Let's assume that there are $F$ blocks to be read from the Hadoop cluster. This means we need to read $\frac{F}{k}$ data strips. If we assume that there is an average of $f_b$ block failures per strip, then $f_b \cdot \frac{F}{k}$ blocks are degraded, where $f_b \leq (n-k)$. Since these blocks are recovered independently during the degraded read process, every read request to a failed block requires downloading of $k$ available data blocks. It takes $\frac{k \cdot B}{\gamma}$ seconds to download $k$ available blocks to repair one block and each unavailable block requires $t_f$ time for unavailability detection by the client. Then the time required for $N_c$ HDFS clients (all clients operate in parallel with each other) to read $F$ blocks each is

$$\underbrace{\frac{N_c \cdot B}{\gamma}\left(F - f_b \cdot \frac{F}{k}\right)}_{\text{Regular Read}} + \underbrace{\frac{F}{k} \cdot f_b^2 \cdot t_f}_{\text{Failure Identification}} + \underbrace{\frac{N_c \cdot B \cdot \left(f_b \cdot \frac{F}{k}\right) \cdot k}{\gamma}}_{\text{Degraded Read}} \tag{5.3}$$

In CoARC, the redundant data download is eliminated, because all unavailable data blocks in a strip are recovered by single degraded read, and the block failures need to be detected only once. This reduces the total time required by $N_c$ HDFS clients to

$$\underbrace{\frac{B}{\gamma}\left(N_c \cdot F - f_b \cdot \frac{F}{k}\right)}_{\text{Regular Read}} + \underbrace{\frac{F}{k} \cdot f_b \cdot t_f}_{\text{Failure Identification}} + \underbrace{\frac{B \cdot F}{\gamma}}_{\text{Degraded Read}} \tag{5.4}$$

Let us consider a typical scenario as mentioned before with $t_f$ = 25 seconds, $F$ = 36 blocks, $N_c$ = 5 clients, and $f_b$ = 3 block failures per strip, where each strip is encoded by using $(10, 6)$ erasure code, i.e, $k$ = 6. Using above equations, HDFS-RAID requires around 1665 seconds, while CoARC requires around 549 seconds only. Out of this, HDFS-RAID spends around 1350 seconds for failure identification, while CoARC spends merely 450 seconds. It can also be observed that HDFS-RAID requires reading of 630 blocks across

the network, while CoARC reads just 198 blocks. This shows that CoARC has a huge potential to reduce both program execution time and network consumption.

### 5.3 Experimental Evaluation

To validate the practicality and effectiveness of CoARC, we prototype it in an HDFS cluster testbed. We implement it on HDFS release 0.22.0 and HDFS-RAID layer [3, 65]. We mainly extend the DFSInputStream to support the data recovery and caching during the degraded read process. We aim to show the actual improvement CoARC has over the traditional degraded mechanism by capturing program execution time and network traffic in the presence of node failures.

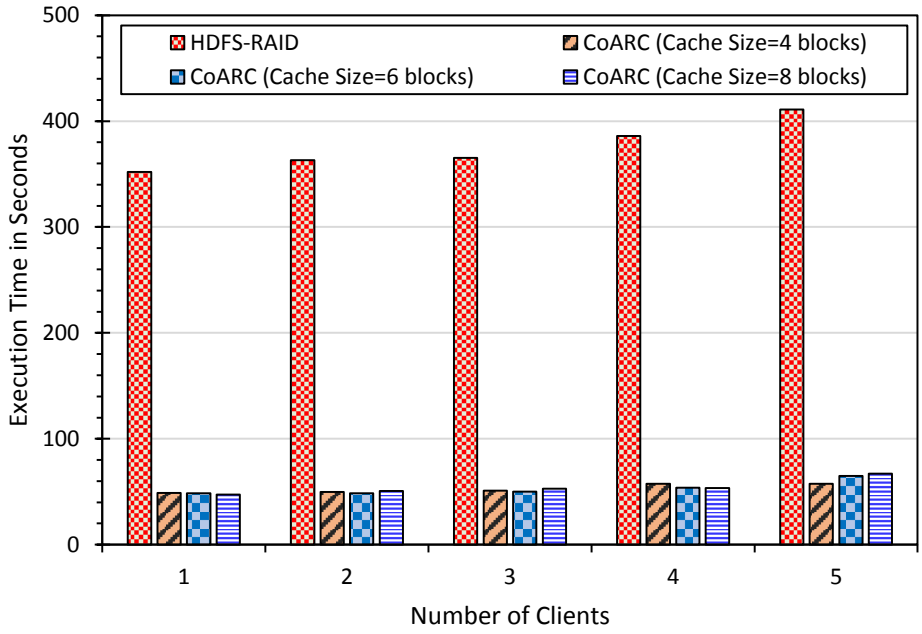



Fig. 48. Total execution time in the Hadoop cluster with triple node-failures. All clients run in parallel. Failures last more than the program execution time to mimic permanent failures.

We deploy an erasure coded Hadoop cluster with 11 DataNodes and 1 NameNode. The RaidNode is instantiated to perform erasure coding, and we use 10 DataNodes to serve

client and reserve one DataNode to use as a CacheNode. The reason for using the DataNode as the cache is because we want the cached data to be persistent with cluster restart. We randomly choose multiple data nodes to run DFS clients for multi-client scenarios. All nodes are Linux-based machines, which contain two 2.30Ghz Intel E52630 processors. Each of the machines has 64GB RAM and 320GB hard drives. For interconnection between nodes, each node is equipped with an Ethernet interface card with a network speed of 1Gbps. All of these physical entities are connected over a 24-port HP 1810-24G switch. We use RS(10,6) code as a representative erasure code. Since our implementation is independent of erasure code, CoARC can be used with any erasure code.

### 5.3.1 Multi-Client Performance

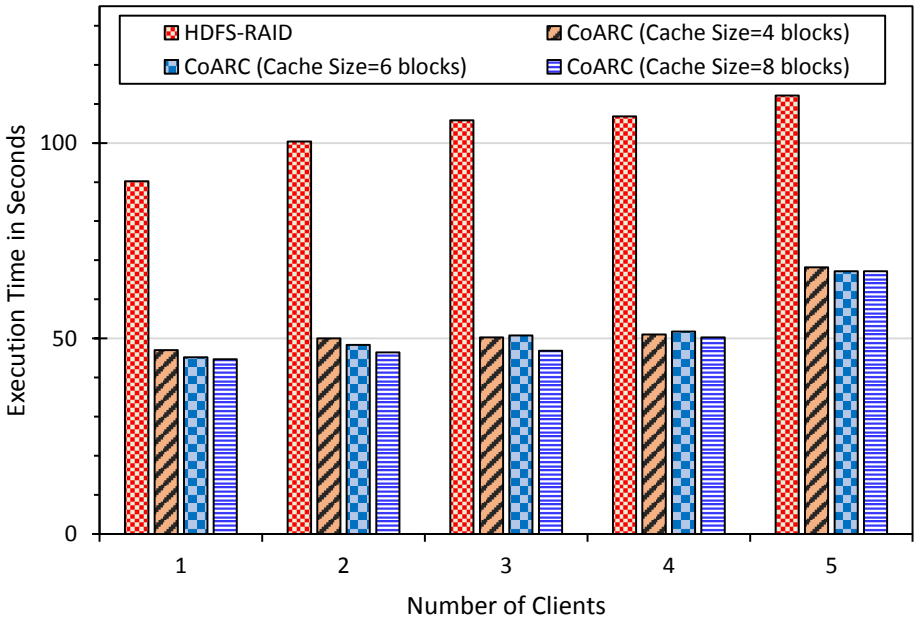



Fig. 49. Total execution time in the Hadoop cluster with 3 node-failures. All clients run in parallel. Failures last for 1 minute only.

Here, we evaluate the impact of the number of clients in the total execution time of a file read. The file size for our experiments is set to span two strip length. Since the

block size used in out HDFS-cluster is 64 MB, we perform multi-client reads on a file of size 768 MB. We introduce 3 node-failures, such that each strip has three unavailable blocks. Figure 48 shows the total execution time for HDFS-RAID and CoARC with various cache sizes. In this test, the failures last longer than program execution time, i.e., from a program's perspective such kinds of failures can be classified as permanent failures. We also vary the cache size of CoARC. We observed that HDFS-RAID spent most of its time in failure detection, thus achieving 350 seconds. On the contrary, CoARC spends much less time for failure detection, which in turn reduces the program execution time to $50 - 60$ seconds. We can see that with the increase in the number of clients, the execution time for HDFS-RAID increases, which is due to more disk interference as all clients try to read the same data blocks during normal operation and degraded reads. In CoARC, since recovery is performed by a single client, the execution time remains fairly constant. We observed that the change in cache size did not have a huge impact on the execution time because the cache in CoARC is an elastic cache and cache eviction is performed in batch every 1 minute. We observed that CoARC could achieve up to $86\%$ improvement of the time spent in performing degraded reads.

Figure 49 shows that when the block unavailability period is less than the program execution time, the performance gain of CoARC is only about $40\%$. When inaccessible data nodes become active later, blocks in those nodes need to go through failure detection, and degraded reads are not required for alive blocks. This causes HDFS-RAID to perform failure detection for fewer blocks, in comparison to the failures lasting for whole program execution time. In the case of CoARC, the execution time is similar to Figure 48 because all data block recoveries are performed in the beginning, and the failure detection is required to be performed only once.

To further validate our claim of reduction in network traffic, we monitor the amount of data reads performed by the clients during the file read. Figure 50 and 51 illustrate the total

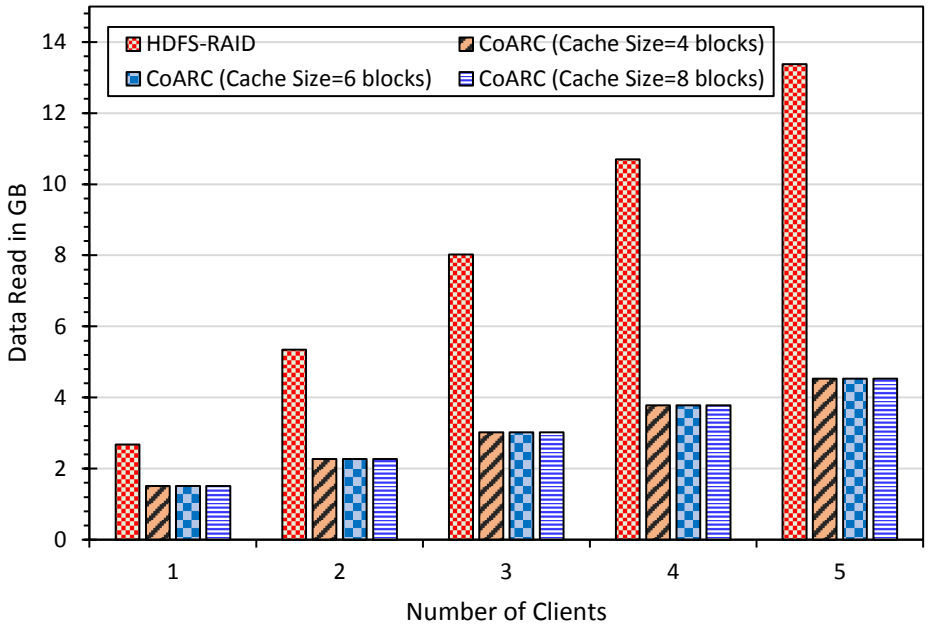



Fig. 50. Total read traffic through the network in the Hadoop cluster with 3 node-failures. All clients run in parallel. Failures last more than the program execution time to mimic permanent failures.

data read from Hadoop cluster for a single file copy from HDFS to the local file system. We can see that there is a linear increase in the data read with the increase in the number of clients. For a file size of $768$ MB with 6 data blocks lost, HDFS-RAID transferred up to $13.4$ GB of data, while CoARC reduces that to $4.5$ GB, which is around $66\%$ reduction in recovery traffic. Since the network is still a scarce resource in cloud environments [59], [55] , CoARC is a viable option for large clusters, where failures are common.

In Figure 51, we see that for a single client, we require more network resources than HDFS-RAID. Although some data blocks come alive later and there is lower network usage for a single client, CoARC still requires less data read while multiple clients are performing reads. It is worth nothing that although the network consumption is less for node failures lasting less than one minute, the execution time is still more than that of CoARC (see Figure 49). This large execution time for HDFS-RAID came from the idling period for

failure detection.

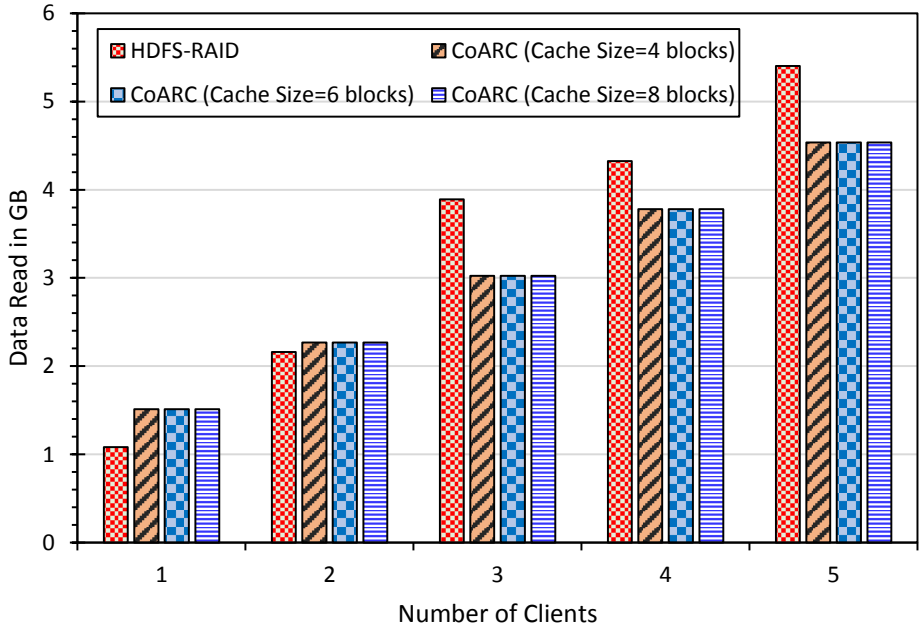



Fig. 51. Total read traffic through the network in the Hadoop cluster with 3 node-failures. All clients run in parallel. Failures last for 1 minute only.

### 5.3.2 MapReduce Workloads

We also evaluate the performance of MapReduce jobs in HDFS-RAID and CoARC in the presence of failures. For our MapReduce applications, we create a dataset of size 768 MB, which contains a collection of English Novels from the Project Gutenberg. We use three MapReduce applications: *TeraSort*, *WordCount*, and *Grep*. TeraSort sorts the data of size 768 MB generated by TeraGen program included with HDFS distribution. WordCount computes the frequency of occurrence of each word in the dataset and Grep extracts the matching word/string from the dataset and counts its occurrence.

Figure 52 shows the total time it took for these MapReduce applications to complete their executive tasks in presence of multi-node failure. We observed that there is a linear increase in the execution time with the increase in node failures, for all the applications

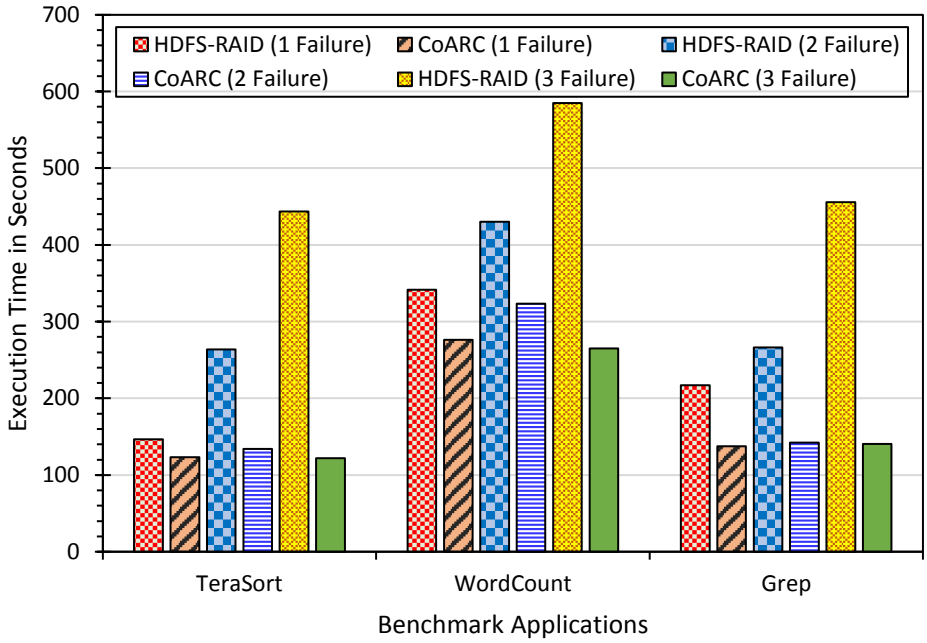



Fig. 52. Total execution time for different MapReduce applications with different failure scenarios. Failures last more than application execution time.

running in HDFS-RAID. In CoARC, this execution time was relatively constant because of non-redundant failure detection and less data traffic. CoARC reduced the execution time by $72.5\%$, $54.6\%$, and $69.1\%$, respectively for TeraSort, WordCount, and Grep applications for triple node failure. It is worth noting that CoARC just reduces the Map time of the MapReduce work flow because the degraded read happen only during the Map phase. Since WordCount spends more time in Reduce phase, we see less performance improvement t han Grep(in terms of reduction in total execution time), even if both workloads perform mapping of the same set of data. We see more gain for TeraSort because it first needs to rebuild the data records from data blocks and then performs the data sort, but the data blocks are unavailable. While block failure identification and data recovery is performed twice during record building phase and Map stage in HDFS-RAID, CoARC eliminates this redundancy.

Figure 53 shows the amount of data reads for these applications. We can clearly

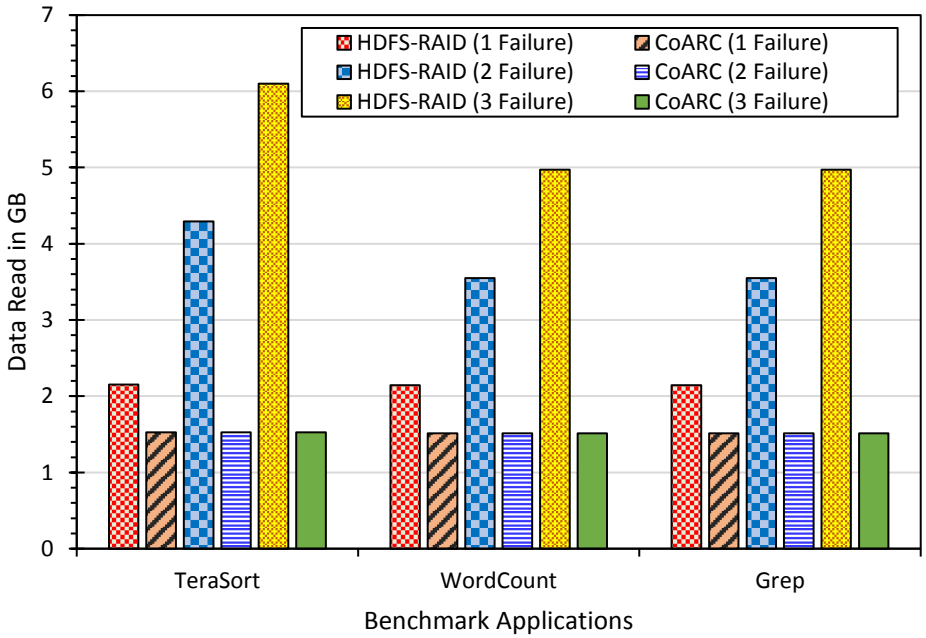



Fig. 53. Total data read traffic for different MapReduce applications with different failure scenarios. Failures last more than application execution time.

see that for these applications, the data read in CoARC is relatively constant despite the number of failures, while it increases significantly in case of HDFS-RAID. For these three applications, CoARC reduced the network consumption by up to $74.9\%$, $69.5\%$, and $69.5\%$. The reason for TeraSort's more network consumption in HDFS-RAID is the twice recovery of some of the unavailable blocks, i.e., once during rebuilding the record and another during Map stage. In CoARC, the unavailable data block is rebuilt during this data record access, and thus has the same network resource usage w.r.t. WordCount and Grep.

For MapReduce applications, CoARC can reduce the network usage even in the presence of short-duration (like 1 minute) failures. In Figure 55, we can easily see that although the data blocks become available later, there happens degraded read in most cases and there is no co-operative recovery among the failures, i.e., all blocks are recovered independently, leading to high network usage in comparison to CoARC. Specifically, in the case of short-duration failures or temporary failures (from the application's perspective), CoARC can

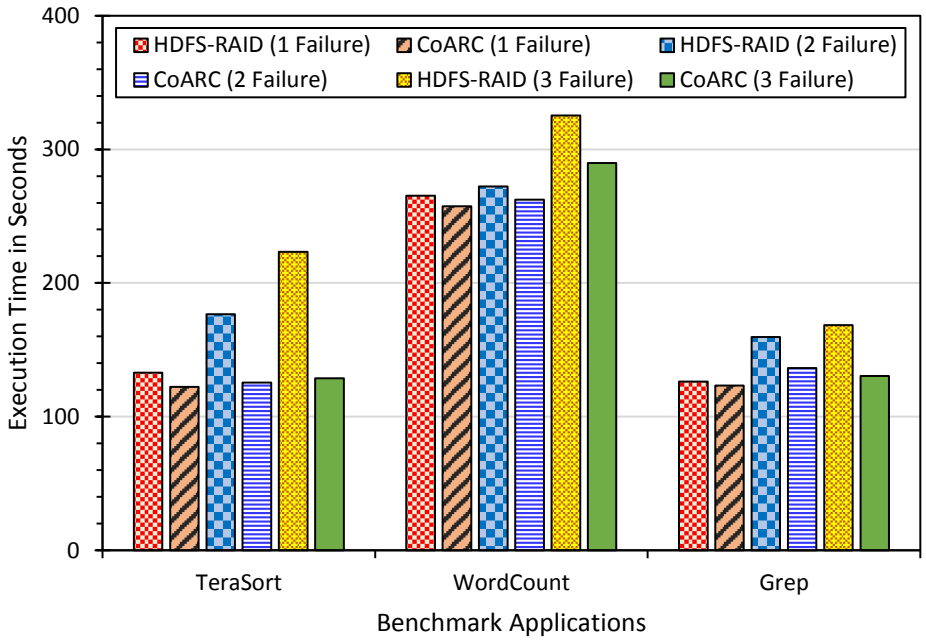



Fig. 54. Total execution time for different MapReduce applications with different failure scenarios. Failures last for 1 minute only.

reduce network traffic by up to $42.4\%$ for TeraSort, $22.7\%$ for WordCount, and $40\%$ for Grep.

We can see that WordCount has a longer execution time, and in the presence of temporary block unavailability, all the data blocks will be available before the program completes. This leads to smaller reduction in the execution time, as seen in Figure 54. CoARC is able to reduce the total execution time by up to $42.3\%$ for TeraSort, $10.9\%$ for WordCount, and $22.6\%$ for Grep in case of triple node failures. One of the things to note is that, in MapReduce workloads, the Map tasks are scheduled based on the unoccupied map slots and if there are enough map slots available for all blocks, all the unavailable blocks will incur degraded reads, leading to high network usage. Although the network usage is large due to recurring reads, these unavailable blocks are accessed in parallel, causing them to share the failure detection time and thus the impact of redundant failure identification will be low.

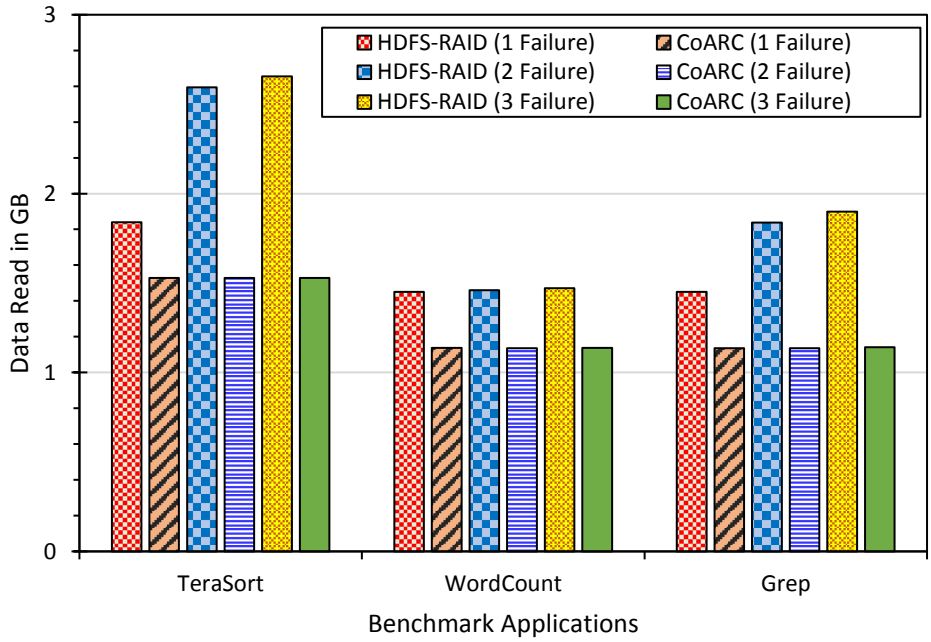



Fig. 55. Total data read traffic for different MapReduce applications with different failure scenarios. Failures last for 1 minute only.

### 5.4 Summary

Failures are the norm in today's data centers. These failures (temporary/permanent) affect the block availability, and degraded reads have become a critical operation in storage systems. Before initiating a degraded read, it is necessary to identify that failure detection time for unavailable blocks plays a significant role in overall application performance. Additionally, it is important to note that when performing conventional degraded reads the system needs to identify the location of all unavailable blocks in that specific data strip, and the amount of data read for $n - k$ failure recovery is the same as single failure recovery for $(n, k)$ MDS codes, like RS code. Our CoARC system, which enhances the degraded read to recover all blocks, with anticipation of future or multi-client access, allows us to eliminate the redundant failure identification of the unavailable blocks and heavily reduce the network traffic incurred due to redundant degraded reads. We implement and verify that

CoARC can reduce the execution time by up to $86\%$ for multi-client file copy operations and up to $72.5\%$ for MapReduce applications. CoARC was also able to reduce the network traffic by up to $42.4\%$ for MapReduce applications and up to $66\%$ for multi-client file reads.

# CHAPTER 6

# CONCLUSIONS

In this dissertation, we made following contributions to improve the reliability, performance, and interoperability of erasure coded storage systems:

1. In the evaluation work proposed in Chapter 2, we explore various erasure codes tolerating three or more failures. We comprehensively compare and evaluate the erasure codes based on the encoding, decoding and rebuild/reconstruction efficiency. We show that there is a trade-off between various parameters like reconstruction chain, code stability, and complexity of the codes and provide some insights into the choice of erasure codes in erasure coded storage systems.

2. In the research work presented in Chapter 3, we apply the concept of erasure code from Chapter 2 in the device layer to improve the SSD's performance and reliability by proposing (EECC) mechanism. This mechanism further improves the performance of erasure coded storage by targeting the read intensive applications and underlying SSD devices. We identify that most of the bit errors can be tolerated by weak-ecc to improve the fault-free operation and apply pipelining concept to further achieve improvement in the read performance. We use Row-Diagonal Parity erasure code to improve the chip level reliability of the SSD.

3. We then propose a new method of performing erasure coding (FINGER) for the widely popular distributed storage system, i.e., Hadoop in Chapter 4 . The FINGER is able to improve both the write and update performance of Hadoop Distributed File System. It is able to perform fine grained erasure coding to eliminate redundant

parity update problem, when performing whole block updates in Hadoop. Thus, we explore the application of the erasure codes into the file system layer for achieving the improvement in system write/update performance, without increasing the file system's metadata size. FINGER is able to improve the interoperability of Hadoop by supporting low latency write and update operations because Hadoop is originally designed for write once, read many applications.

4. The CoARC mechanism presented in Chapter 5 targets the improvement in the reliability and recovery performance of Hadoop. CoARC is a novel way of performing the recovery of data during temporary failures. It co-operatively identifies and recovers the data block failures and then caches the data for efficient data sharing among various MapReduce applications and clients. We also present a novel cache algorithm for recovery caches in erasure coded file systems. The CoARC also improves the interoperability by proving a mechanism to share recovered data across various applications and clients.

5. We analyze and evaluate the impact of erasure codes in storage systems, and design various new techniques by applying the erasure coding concepts in various layers of storage systems. To demonstrate the effectiveness of EECC, FINGER, and COARC for achieving gain in reliablity, performance and interoperability, we perform analysis, simulation and real-world implementations.

## LIST OF PUBLICATIONS

This dissertation is mainly based on the following papers:

- **P. Subedi**, P. Huang, T. Liu, J. Moore, S. Skelton, and X. He, "CoARC: Co-operative, Aggressive Recovery and Caching for Failures in Erasure Coded Hadoop", In *Proc. of the* $45^{th}$ *International Conference on Parallel Processing (ICPP'16)*, Aug 2016.

- **P. Subedi**, P. Huang, B. Young, and X. He, "FINGER: A Novel Erasure Coding Scheme Using Fine Granularity Blocks to Improve Hadoop Write and Update Performance", In *Proc. of the* $10^{th}$ *IEEE International Conference on Networking, Architecture, and Storage (NAS'15)*, Aug 2015.

- **P. Subedi**, P. Huang, X. He, M. Zhang, and J. Han, "A Hybrid Erasure-Coded ECC Scheme to Improve Performance and Reliability of Solid State Drives", In *Proc. of the* $33^{rd}$ *IEEE International Performance Computing and Communications Conference (IPCCC'14)*, December 2014. **Best Paper Runner-up**.

- **P. Subedi** and X. He, "A Comprehensive Analysis of XOR-based Erasure Codes Tolerating 3 or More Concurrent Failures". In *Proc. of the* $18^{th}$ *IEEE Workshop on Dependable Parallel, Distributed and Network-Centric Systems (DPDNS), held with IPDPS2013*, May 2013

# REFERENCES


[1] Amazon Simple Storage Service. http://aws.amazon.com/s3/.

[2] *HBase: The Definitive Guide*.

[3] HDFS-RAID. http://wiki.apache.org/hadoop/HDFS-RAID.

[4] MapR Distribution. https://www.mapr.com/products/mapr-distribution-including-apache-hadoop.

[5] Transient Fault Handling: Patterns and Practices. https://msdn.microsoft.com/en-us/library/hh675232.aspx.

[6] Every Day We Create 2.5 Quintillion Bytes of Data. http://www.storagenewsletter.com/rubriques/market-reportsresearch/ibm-cmo-study/, 2011.

[7] M. Abd-El-Malek, W. V. Courtright, II, C. Cranor, G. R. Ganger, J. Hendricks, A. J. Klosterman, M. Mesnier, M. Prasad, B. Salmon, R. R. Sambasivan, S. Sinnamohideen, J. D. Strunk, E. Thereska, M. Wachs, and J. J. Wylie. Ursa Minor: Versatile Cluster-based Storage. In *Proceedings of the* $4^{th}$ *USENIX Conference on File and Storage Technologies*, 2005.

[8] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy. Design Tradeoffs for SSD Serformance. In *Proceedings of the USENIX 2008 Annual Technical Conference on Annual Technical Conference*, ATC, 2008.

[9] M. Blaum, J. Bruck, and A. Vardy. MDS Array Codes with Independent Parity Symbols. *IEEE Transactions on Information Theory*, 1996.

[10] M. Blaum et al. EVENODD: An Efficient Scheme for Tolerating Double Disk Failures in RAID Architectures. *IEEE Transactions on Computers*, 1995.

[11] M. Blaum and R. Roth. On Lowest Density MDS Codes. *IEEE Transactions on Information Theory*, 1999.

[12] J. Blomer et al. An XOR-based Erasure-Resilient Coding Scheme. Technical report, International Computer Science Institute, 1995.

[13] J. S. Bucy and G. R. Ganger. The DiskSim Simulation Environment Version 3.0 Reference Manual. Technical report, 2003.

[14] Y. Cai, G. Yalcin, O. Mutlu, E. Haratsch, A. Cristal, O. Unsal, and K. Mai. Flash Correct-and-Refresh: Retention-aware Error Management for Increased Flash Memory Lifetime. In *Proceedings of the 30th International Conference on Computer Design (ICCD)*, 2012.

[15] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, J. Haridas, C. Uddaraju, H. Khatri, A. Edwards, V. Bedekar, S. Mainali, R. Abbasi, A. Agarwal, M. F. u. Haq, M. I. u. Haq, D. Bhardwaj, S. Dayanand, A. Adusumilli, M. McNett, S. Sankaran, K. Manivannan, and L. Rigas. Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency. In *Proceedings of the* $23^{rd}$ *ACM Symposium on Operating Systems Principles*, 2011.

[16] S. Caron, F. Giroire, D. Mazauric, J. Monteiro, and S. Pérennes. Data Life Time for Different Placement Policies in P2P Storage Systems. In $3^{rd}$ *International Conference on Data Management in Grid and Peer-to-Peer Systmes*, 2010.

[17] Y. Cassuto and J. Bruck. Low-Complexity Array Codes for Random and Clustered 4-Erasures. *IEEE Transactions on Information Theory*, 2012.

[18] A. M. Caulfield, L. M. Grupp, and S. Swanson. Gordon: Using Flash Memory to Build Fast, Power-efficient Clusters for Data-intensive Applications. In *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*, 2009.

[19] J. C. W. Chan, Q. Ding, P. P. C. Lee, and H. H. W. Chan. Parity Logging with Reserved Space: Towards Efficient Updates and Recovery in Erasure-coded Clustered Storage. In *Proceedings of the* $12^{th}$ *USENIX Conference on File and Storage Technologies*, 2014.

[20] Y.-B. Chang and L.-P. Chang. A Self-balancing Striping Scheme for NAND-Flash Storage Systems. In *Proceedings of the ACM symposium on Applied computing*, 2008.

[21] P. Chen, E. Lee, et al. RAID: High-Performance, Reliable Secondary Storage. *ACM Computing Surveys*, 1994.

[22] P. Corbett, B. English, A. Goel, T. Grcanac, S. Kleiman, J. Leong, and S. Sankar. Row-Diagonal Parity for Double Disk Failure Correction. In *Proceedings of the* $3^{rd}$ *USENIX Symposium on File and Storage Technologies*, 2004.

[23] A. Dan and D. F. Towsley. An Approximate Analysis of the LRU and FIFO Buffer Replacement Schemes. In *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 1990.

[24] J. Dean. Experiences with MapReduce, an Abstraction for Large-scale Computation. In *Proceedings of the* $15^{th}$ *International Conference on Parallel Architectures and*

*Compilation Techniques*, pages 1–1, 2006.

[25] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 2008.

[26] A. G. Dimakis, B. Godfrey, Y. Wu, M. J. Wainwright, and K. Ramchandran. Network Coding for Distributed Storage Systems. *IEEE Transactions on Information Theory*, 2010.

[27] K. S. Esmaili, L. Pamies-Juarez, and A. Datta. CORE: Cross-object Redundancy for Efficient Data Repair in Storage Systems. In *Proceedings of the IEEE International Conference on BigData*, 2013.

[28] B. Fan, W. Tantisiriroj, L. Xiao, and G. Gibson. DiskReduce: RAID for Data-intensive Scalable Computing. In *Proceedings of the* $4^{th}$ *Annual Workshop on Petascale Data Storage*, 2009.

[29] G. L. Feng, R. H. Deng, F. Bao, and J. C. Shen. New Efficient MDS Array Codes for RAID Part I: Reed-Solomon-Like Codes for Tolerating Three Disk Failures. *IEEE Transactions on Computers*, 2005.

[30] G. L. Feng, R. H. Deng, F. Bao, and J. C. Shen. New Efficient MDS Array Codes for RAID part II: Rabin-Like Codes for Tolerating Multiple (greater than or equal to 4) Disk Failures. *IEEE Transactions on Computers*, 2005.

[31] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. *SIGOPS Operating Systems Review*, 2003.

[32] K. Greenan, D. D. E. Long, E. L. Miller, T. Schwarz, and A. Wildani. Building Flexible, Fault-Tolerant Flash-based Storage Systems. In *Proceedings of the* $5^{th}$ *Workshop on Hot Topics in System Dependability*, 2009.

[33] A. Gupta, Y. Kim, and B. Urgaonkar. DFTL: A Flash Translation Layer Employing Demand-based Selective Caching of Page-level Address Mappings. *ACM SIGPLAN Notices*, 2009.

[34] A. Haeberlen, A. Mislove, and P. Druschel. Glacier: Highly Durable, Decentralized Storage Despite Massive Correlated Failures. In *Proceedings of the* $2^{nd}$ *Symposium on Networked Systems Design and Implementation*, 2005.

[35] J. Hafner. WEAVER Codes: Highly Fault Tolerant Erasure Codes for Storage Systems. In *Proceedings of the USENIX Conference on File and Storage Technologies*, 2005.

[36] J. Hafner. HoVer Erasure Codes For Disk Arrays. In *Proceedings of the International Conference on Dependable Systems and Networks*, 2006.

[37] R. W. Hamming. Error Detecting and Error correcting codes. *Bell System Technical Journal*, 1950.

[38] T. Harter, D. Borthakur, S. Dong, A. Aiyer, L. Tang, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Analysis of HDFS Under HBase: A Facebook Messages Case Study. In *Proceedings of the* $12^{th}$ *USENIX Conference on File and Storage Technologies*, 2014.

[39] P. Heidelberger, J. K. Muppala, and K. S. Trivedi. Accelerating Mean Time to Failure Computations. *Performamce Evaluation*, 1996.

[40] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin. Erasure Coding in Windows Azure Storage. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, 2012.

[41] C. Huang and L. Xu. STAR: An Efficient Coding Scheme for Correcting Triple Storage Node Failures. In *Proceedings of the USENIX Conference on File and Storage Technologies*, 2005.

[42] S. Im and D. Shin. Flash-Aware RAID Techniques for Dependable and High-Performance Flash Memory SSD. *IEEE Transactions on Computers*, 2011.

[43] Intel. Intel RAID Controller RS25DB080. http://www.intel.com/content/www/us/en/servers/raid/raid-controller-rs25db080.html.

[44] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed Data-parallel Programs from Sequential Building Blocks. In *Proceedings of the* $2^{nd}$ *ACM SIGOPS/EuroSys European Conference on Computer Systems*, 2007.

[45] W. K. Josephson, L. A. Bongo, K. Li, and D. Flynn. DFS: A File System for Virtualized Flash Storage. *Transactions on Storage*, 2010.

[46] O. Khan, R. Burns, J. Plank, W. Pierce, and C. Huang. Rethinking Erasure Codes for Cloud File Systems: Minimizing I/O for Recovery and Degraded Reads. In *Proceedings of the* $10^{th}$ *USENIX Conference on File and Storage Technologies*, 2012.

[47] M. Kryder and C. S. Kim. After Hard Drives-What Comes Next? *IEEE Transactions on Magnetics*, 45(10), 2009.

[48] S.-W. Lee, W.-K. Choi, and D.-J. Park. FAST: An Efficient Flash Translation Layer for Flash Memory, 2006.

[49] Y. Lee, S. Jung, and Y. H. Song. FRA: A Flash-aware Redundancy Array of Flash Storage Devices. In *Proceedings of the 7th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis*, 2009.

[50] R. Li, P. P. C. Lee, and Y. Hu. Degraded-First Scheduling for MapReduce in Erasure-Coded Storage Clusters. In $44^{th}$ *Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2014.

[51] F. MacWilliams and N. Sloane. *The Theory of Error-Correcting Codes*. Northholland Publishing Company, 2nd edition, 1978.

[52] N. Mielke, T. Marquart, N. Wu, J. Kessenich, H. Belgal, E. Schares, F. Trivedi, E. Goodness, and L. Nevill. Bit Error Rate in NAND Flash Memories. In *Proceedings of* $46^{th}$ *IEEE International Reliability Physics Symposium*, 2008.

[53] B. Nicolae, G. Antoniu, L. Bougé, D. Moise, and A. Carpen-Amarie. BlobSeer: Next-generation Data Management for Large Scale Infrastructures. *Journal of Parallel and Distributed Computing*, 71(2).

[54] A. One. YAFFS: Yet Another Flash File System. http://www.yaffs.net/, 2002.

[55] L. Pamies-Juarez, F. Blagojević, R. Mateescu, C. Gyuot, E. E. Gad, and Z. Bandić. Opening the Chrysalis: On the Real Repair Performance of MSR Codes. In $14^{th}$ *USENIX Conference on File and Storage Technologies*, 2016.

[56] D. Patterson, G. Gibson, and R. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proceedings of the ACM SIGMOD/PODS Conference*, 1988.

[57] J. S. Plank. A Tutorial on Reed-Solomon Coding for Fault-Tolerance in RAID-Like Systems. *Software - Practice and Experience*, 1997.

[58] M. Rabin. Efficient Dispersal of Information for Security, Load Balancing, and Fault Tolerance. *Journal of the ACM*, 1989.

[59] B. T. Rao, N. V. Sridevi, V. K. Reddy, and L. S. S. Reddy. Performance Issues of Heterogeneous Hadoop Clusters in Cloud Computing. *CoRR*, 2012.

[60] I. Reed and G.Solomon. Polynomial Codes Over Certain Finite Fields. *Journal of the Society for Industries and Applied Mathematics*, 1960.

[61] J. T. Robinson and M. V. Devarakonda. Data Cache Management Using Frequency-Based Replacement. In *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 1990.

[62] M. Sathiamoorthy, M. Asteris, D. Papailiopoulos, A. G. Dimakis, R. Vadali, S. Chen, and D. Borthakur. XORing Elephants: Novel Erasure Codes for Big Data. In *Proceedings of the* $39^{th}$ *International Conference on Very Large Data Bases*, 2013.

[63] E. Schneider. Method, Product, and Apparatus for Processing a Data Request, 2015.

[64] M. Schulze, G. Gibson, R. Katz, and D. Patterson. How Reliable is a RAID? In *Digest of Papers* $34^th$ *IEEE Computer Society International Conference: Intellectual Leverage*, 1989.

[65] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop Distributed File System. In *Proceedings of the* $26^{th}$ *Symposium on Mass Storage Systems and Technologies*, 2010.

[66] G. Soundararajan, V. Prabhakaran, M. Balakrishnan, and T. Wobber. Extending SSD Lifetimes with Disk-based Write Caches. In *Proceedings of the 8th USENIX conference on File and Storage Technologies*, 2010.

[67] T. Steinke, K. Peter, and S. Borchert. Efficiency Considerations of Cauchy Reed-Solomon Implementations on Accelerator and Multi-Core Platforms. *Proceedings of the Symposium in Application Accelerators in High Performance Computing*, 2010.

[68] Storage Performance Council. SPC Trace File Format Specification. http://traces.cs.umass.edu/index.php/Storage/Storage, 2010.

[69] F. Sun, K. Rose, and T. Zhang. On the Use of Strong BCH Codes for Improving Multilevel NAND Flash Memory Storage Capacity. In *Proceedings of the IEEE Workshop on Signal Processing Systems (SiPS): Design and Implementation*, 2006.

[70] D. Tang, X. Wang, S. Cao, and Z. Chen. A New Class of Highly Fault Tolerant Erasure Code for Disk Array. *Workshop on Power Electronics and Intelligent Transportation Systems*, 2008.

[71] C. Tau and T. Wang. Efficient Parity Placement Schemes for Tolerating Triple Disk Failures in RAID Architectures. *International Conference on Advanced Information Networking and Applications*, 2003.

[72] J. Tian, Z. Yang, W. Chen, B. Zhao, and Y. Dai. Probabilistic Failure Detection for Efficient Distributed Storage Maintenance. In *IEEE Symposium on Reliable Distributed Systems*, 2008.

[73] Y. Wang and G. Li. Rotary-code: Efficient MDS Array Codes for RAID-6 Disk Arrays. *WSEAS Transactions on Computers*, 2009.

[74] Y. Wang, G. Li, and X. Zhong. Triple-Star: A Coding Scheme with Optimal Encoding Complexity for Tolerating Triple Disk Failures in RAID. *International Journal of Innovative Computing, Information and Control*, 2012.

[75] H. Weatherspoon and J. Kubiatowicz. Erasure Coding Vs. Replication: A Quantitative Comparison. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, 2002.

[76] C. Wu, S. Wan, X. He, Q. Cao, and C. Xie. H-Code: A Hybrid MDS Array Code to Optimize Partial Stripe Writes in RAID-6. In *Proceedings of the* $25^{th}$ *International Parallel and Distributed Processing Symposium*, 2011.

[77] G. Wu, B. Eckart, and X. He. BPAC: An Adaptive Write Buffer Management Scheme for Flash-based Solid State Drives. In *Proceedings of the IEEE* $26^{th}$ *Symposium on Mass Storage Systems and Technologies*, 2010.

[78] G. Wu and X. He. Delta-FTL: Improving SSD Lifetime via Exploiting Content Locality. In *Proceedings of the* $7^{th}$ *ACM European Conference on Computer Systems*, 2012.

[79] G. Wu, X. He, N. Xie, and T. Zhang. DiffECC: Improving SSD Read Performance Using Differentiated Error Correction Coding Schemes. In *Proceedings of the IEEE International Symposium on Modeling, Analysis Simulation of Computer and Telecommunication Systems*, 2010.

[80] Q. Xin, E. L. Miller, T. Schwarz, D. D. E. Long, S. A. Brandt, and W. Litwin. Reliability Mechanisms for Very Large Storage Systems. In *Proceedings of the* $20^{th}$ *IEEE/*$11^{th}$ *NASA Goddard Conference on Mass Storage Systems and Technologies*, 2003.

[81] Q. Xin, E. L. Miller, and T. J. E. Schwarz. Evaluation of Distributed Recovery in Large-Scale Storage Systems. In *Proceedings of the* $13^{th}$ *International Symposium on High-Performance Distributed Computing*, 2004.

[82] L. Xu and J. Bruck. X-Code: MDS Array Codes with Optimal Encoding. *IEEE Transactions on Information Theory*, 1999.

[83] Q. Yang and J. Ren. I-CASH: Intelligently Coupled Array of SSD and HDD. In *17th International Symposium on High Performance Computer Architecture (HPCA)*, 2011.

[84] Z. Zhang, A. Deshpande, X. Ma, E. Thereska, and D. Narayanan. Does Erasure Coding Have A Role To Play In My Data Center. 2010.

[85] K. Zhao, W. Zhao, T. Zhang, X. Zhang, and N. Zeng. LDCP-in-SSD: Making Advanced Error Correction Codes Work Effectively in Solid State Drives. Proceedings of the USENIX Conference on File and Storage Technologies, 2013.

[86] Y. Zhu, J. Lin, P. P. C. Lee, and Y. Xu. Boosting Degraded Reads in Heterogeneous Erasure-Coded Storage Systems. *IEEE Transactions on Computers*, 2015.

# VITA

Pradeep Subedi was born in Feb. 12, 1988 in Pokhara, Nepal. He graduated from SOS Herman Gmeiner School Gandaki in 2005. He received his Bachelors in Electronics and Communication Engineering from Institute of Engineering, Pulchowk Campus (Tribhuvan University), Lalitpur, Nepal in 2010. He is also the recipient of 2016 ECE Outstanding Graduate Teaching Assistant Award from School of Engineering, USENIX Student Travel Grant, and NAS Student Travel Grant.